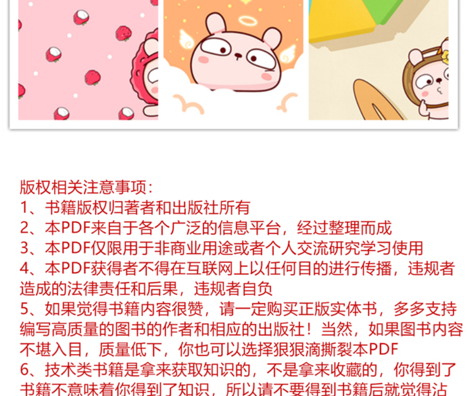


版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF



仅供非商业用途或交流学习使用

微服务实践

Practical Microservices

[印度] 乌姆什·拉姆·夏尔玛（Umesh Ram Sharma）著

占红来 刘博 译

人 民 邮 电 出 版 社

北 京

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

微服务实践 / (印) 乌姆什·拉姆·夏尔玛 (Umesh Ram Sharma) 著 ; 占红来, 刘博译. — 北京 : 人民邮电出版社, 2019.1

书名原文: Practical Microservices

ISBN 978-7-115-49870-0

Ⅰ. ①微… Ⅱ. ①乌… ②占… ③刘… Ⅲ. ①互联网络—网络服务器 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字(2018)第252033号

版权声明

Copyright ©2017 Packt Publishing. First published in the English language under the title *Practical Microservices*, ISBN 978-1-78588-508-2. All rights reserved.

本书中文简体字版由 Packt Publishing 公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

◆ 著 [印度] 乌姆什·拉姆·夏尔玛（Umesh Ram Sharma）
译 占红来 刘 博
责任编辑 杨海玲
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市祥达印刷包装有限公司印刷

◆ 开本：800×1000 1/16
印张：13
字数：242 千字 2019 年 1 月第 1 版
印数：1－2 600 册 2019 年 1 月河北第 1 次印刷
著作权合同登记号 图字：01-2017-8624 号

定价：49.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

仅供非商业用途或交流学习使用

内容提要

近些年来，微服务一直是非常热门的话题，关于微服务架构的讨论也是层出不穷。本书以贯穿整书的示例为出发点，由浅入深地阐述使用微服务的最佳实践，以及如何避免采用微服务架构可能带来的复杂性陷阱。本书从微服务架构本身的特征入手，讨论微服务组件的设计指导原则、有效通信的方式以及常见的安全挑战和数据模型的选择；然后进入微服务架构的测试部分，探讨微服务的测试挑战和解决方法、监控和扩展常用的实践以及如何将现有架构演变为微服务架构；最后总结微服务架构在设计和开发方面遇到的常见问题及解决方案。

本书既适合想学习微服务架构的开发人员作为入门参考书，也适合有微服务开发经验的技术人员在实践中遇到问题时作为技术手册，查阅最佳实践和解决方案。

仅供非商业用途或交流学习使用

译者序

2016 年 11 月，我刚刚加入 ThoughtWorks，就有幸与滕云（《实现领域驱动设计》译者）在同一个项目组工作。滕云与我们分享了领域驱动设计与微服务架构实践之间的潜在联系，例如，如何利用领域驱动设计中的边界上下文（bounded context）对微服务进行划分，如何在微服务设计中识别聚合根（aggregate root），等等。这些分享大大提升了我对微服务架构的兴趣，也纠正了自己对微服务理解不正确的地方。

据了解在 ThoughtWorks “毕业”的标准基本上可以归纳为两个字——出书。彼时我就在想，出书挑战很大，那自己可不可以从翻译一本书入手呢？这样一来可以锻炼自己，为出书提前磨炼一下，二来也至少达到了“肄业”的标准。恰巧在 2017 年 9 月，海侠在群里发了很多需要翻译的书名，让大家自己挑选。其中有一本关于微服务的书让我眼前一亮，于是我立刻回信要求翻译。孰料得到回复说已经有人抢先选走了。正在郁闷之际，好消息传来，我的同事占红来愿意与我共同翻译本书，以飨读者。

本书的前 5 章由 ThoughtWorks 资深开发工程师占红来翻译，而我一直活跃在测试领域，因此负责后 5 章的翻译。之所以这样分工，是因为我们想在各自熟悉的领域准确地理解作者想表达的意图，力求尽可能地贴近原文进行翻译表述。在翻译过程中，尽管我们一丝不苟地进行审查和校对，但仍难免有所疏漏。如果读者在阅读过程中发现某些瑕疵或有争议的地方，非常欢迎联系我们，以便我们进行核对和更正，避免对大家造成困扰和误导。我的联系方式是 hitliubo@gmail.com。

感谢我的同事占红来无私分享了此次翻译的机会，并以翻译界资深人士的身份带我迈进了翻译的大门。感谢他在忙碌的工作之余，指导我共同完成翻译及校对工作。

感谢人民邮电出版社资深高级策划编辑杨海玲女士的大力支持与宽容，她不仅容忍了我们的终极拖延症，还在我们遇到项目安排与翻译时间冲突时给予灵活的调节和帮助。

感谢我的爱人艾华和儿子小午，是你们一直以来的理解和支持才让本书的翻译工作得以完成。

刘博

2018 年 6 月

于西安 ThoughtWorks 办公室

仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

译者简介

占红来是一位咨询师，致力于帮助客户和成就客户。曾主持过某世界 500 强等大型公司的软件一体化开发平台的测试能力提升等落地项目，得到客户的一致好评，并受邀再次合作。

刘博毕业于哈尔滨工业大学，是一位拥有十多年测试经验的软件工程师，主攻自动化测试、性能测试和架构调优领域，对这些领域流行的技术体系和架构风险都有准确的把握。他积极参加对外的技术论坛，把在 IBM、活跃网络和思特沃克的经验积累加以总结并分享给业内相关人士，获得众多好评。

仅供非商业用途或交流学习使用

作者简介

Umesh Ram Sharma 是一名软件开发工程师，在可扩展、分布式云服务应用的架构、设计及开发方面有 8 年以上的经验。

他从印度卡纳塔克邦州开放大学获得信息技术专业的硕士学位。出于对微服务和 Spring 的兴趣，他成了 J2EE、JavaScript、Struts、Hibernate 和 Spring 方面的专家，也具有 AWS、J2EE、MySQL、MongoDB、memchached、Apache、Tomcat 和 Hazelcast 等技术的实践经验。

Umesh Ram Sharma 目前是 ZestMoney 公司的首席软件工程师，帮助他的团队将当前项目迁移至微服务。闲暇时，他喜欢开车兜风、烹饪和参加新技术的各种大会。

仅供非商业用途或交流学习使用

审阅者简介

Yogendra Sharma 是一名 Java 软件开发工程师，同时拥有 Python 的开发经验，主要精力集中在中间件开发。他拥有计算机科学技术专业的学士学位。

目前 Yogendra 是物联网云架构师，就职于印度浦那的 Intelizign 工程服务（Intelizign Engineering Services）公司。他经常探索技术方面的新鲜事物，并保持开放心态，渴望学习新技术和新框架。

Yogendra 也作为技术审阅人审阅了 *Mastering Python Design Patterns*、*Test-Driven Development with Django*、*Spring 4.0 Microservices*、*Distributed Computing with Java 9*，以及音像课程 *Python Projects*、*Learning Python Data Analysis*、*Django Projects: E-Learning Portal*、*Low-Level Python Network Attacks*。以上图书或音像课程均由 Packt 出版社出版。

我想感谢我的父母允许我学习我自己感兴趣的东西，还要感谢我的朋友们的大力支持和鼓励。

仅供非商业用途或交流学习使用

前言

在技术世界中，微服务正变得越来越流行，并得到了技术爱好者、开发人员以及许多文章的广泛关注。本书旨在为开发人员提供编写、测试、保护和部署微服务方面的实用指导。使用微服务有很多好处，但也会面临独特的挑战。本书试图以易于理解的方式来阐述使用微服务的最佳实践，同时也阐述了如何避免采用微服务架构可能带来的复杂性陷阱。

各章介绍

第 1 章会介绍微服务架构的整体概念，其中包括对微服务的基本定义。之后会快速浏览本书中所使用的样例程序。

第 2 章将阐明定义微服务组件的指导原则，并说明这些组件如何成为微服务架构的核心支柱。我们会通过 Spring Boot 框架的 Java 项目来有效地定义微服务组件，进而解释这些指导原则如何在实际中使用。最后，通过一个微服务样例来实际演示基于 Java 的微服务组件如何进行配置和服务发现。

第 3 章会讨论并论证微服务间有效通信的原则。之后介绍使用不同技术（如使用 Spring 框架本身或消息代理）实现同步和异步通信的选择标准。我们也会讲到常见问题的最佳处理方法。

第 4 章会讨论使用微服务架构时可能会遇到的常见安全问题和安全挑战。可以通过引入 JWT、OpenID 和 OAuth 2.0 来提高微服务架构的安全性。

第 5 章首先解释基于微服务的数据模型与传统的数据模型的不同之处以及不同的原因。之后会解释数据技术如何被混合使用，以及针对每个微服务组件如何选择适合的数据管理策略。我们还将探索数据模型，解释不同数据模型的选择策略以及选择理由。

第 6 章将探讨测试技术在经常变化和自动部署的系统中的重要性。传统的测试方法和测试标准能完美地匹配微服务架构吗？或者说，我们需要探索完全不同的测试方法来适应微服务架构吗？也许两者都需要。

第 7 章主要着眼于微服务的部署。在微服务架构中，部署操作是非常频繁的，这是

仅供非商业用途或交流学习使用

要把部署操作变得尽可能无痛点并且易于使用的一个原因。另外一个原因是，对自动化的追求和对系统扩展性的期望，会导致经常中断并部署新的微服务。Docker 会帮助我们定义微服务的部署流程，并将其自动化。

第 8 章将描述将系统演进至微服务架构的主要机制。这一章还会以示例说明如何演进至微服务架构。

第 9 章会描述监控和扩展基于微服务的系统的重要概念和准则，还会探索监控和扩展基于 Java 的微服务的实际方法，并给出示例说明如何进行监控和扩展。

第 10 章会总结在设计和开发微服务架构时的常见问题，并阐述常见的解决方法或最优方案。

系统需求

推荐使用 Linux 操作系统运行本书的示例，当然也可以使用 Windows 或 Mac 操作系统。但请注意，本书所描述的各种步骤均是以 Linux 为基础的。除此之外，还需要使用 Maven、Java 8 和任意一个 Java 集成开发环境，如 Eclipse、IntelliJ 或 STS。推荐使用 MySQL 社区版作为数据库。

目标读者

本书的目标读者是想开始使用微服务，并希望在实际工作中实践微服务的 Java 工程师。阅读本书的读者无须具备微服务的任何相关知识。

排版约定

在阅读本书时，读者会发现有些文本的样式与其他信息不同。下面是这些样式的样例及其含义。

本书采用以下格式来表示代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、链接、用户输入的信息和 Twitter 用户名：“浏览至`/src/main/java/com/sample/firstboot/controller` 文件夹，创建 `SampleController.java` 文件。”

代码清单如下所示：

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayExapleInSpring
{
```


仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF


```
    public static void main (String[] args)
    {
        SpringApplication.run(ApiGatewayExampleInSpring.class, args);
    }
}
```

命令行的输入或输出如下所示：

curl http://localhost:8888/userService/default

新术语或**重要字句**会加粗。例如，我们在使用计算机时见到的菜单或弹出框，在文中的提示格式像这样：“点击 **Switch to the full version**（切换至完整版本）按钮来获取本项目的详细信息。”

警告或重要提示。

技巧和提示。

仅供非商业用途或交流学习使用

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供源代码，要获得以上配套资源，请在异步社区本书页面中点击 配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。



仅供非商业用途或交流学习使用

扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“**异步社区**”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“**异步图书**”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区　　微信服务号

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

目录

第 1 章　微服务架构简介……1

1.1　常规微服务架构……2
1.2　微服务架构的特征……2
1.2.1　问题定义……2
1.2.2　解决方案……3
1.3　做好微服务架构面临的挑战……4
1.3.1　通过日志调试……5
1.3.2　服务监控……5
1.3.3　公共库……5
1.3.4　服务之间的消息传递……5
1.3.5　微服务的部署和版本管理……6
1.4　微服务的未来……6
1.4.1　无服务架构……7
1.4.2　微服务即 PaaS……7
1.5　与传统架构相比微服务架构的优势……7
1.6　是不是看起来与 SOA 很像……9
1.7　将业务领域划分为微服务组件……11
1.8　到底要不要使用微服务……13
1.8.1　组织认同度……13
1.8.2　体验 DevOps……14
1.8.3　分析现有数据库模型……14
1.8.4　自动化和 CI/CD……14
1.8.5　集成……14
1.8.6　安全……14
1.8.7　成功迁移的例子……15
1.9　示例项目（信用风险评估引擎）……15
1.10　Spring……15
1.11　Spring Boot……16
1.12　小结……19

第 2 章　定义微服务组件……21

2.1　微服务的定义……21
2.2　服务发现及其用途……22
2.2.1　DNS……22
2.2.2　服务发现的请求……23
2.2.3　服务发现模式示例……26
2.2.4　整个架构中的配置外化……29
2.3　API 网关及其诉求……32
2.3.1　认证鉴权……33
2.3.2　不同协议……34
2.3.3　负载均衡……34
2.3.4　请求转发（包括服务发现）……34
2.3.5　响应转换……34
2.3.6　断路器……35
2.3.7　API 网关的优劣性……35
2.4　API 网关的例子……36
2.5　用户注册微服务的开发……39
2.6　小结……59



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 3 章　微服务端点之间的通信……61
3.1　微服务间应该如何通信……61
3.2　编制和编排……62
3.2.1　编制……62
3.2.2　编排……63
3.3　同步通信和异步通信……65
3.3.1　同步通信……65
3.3.2　异步通信……68
3.3.3　财务服务……79
3.4　小结……90
第 4 章　微服务端点的安全……91
4.1　微服务的安全挑战……91
4.1.1　复合技术栈或者存在遗留代码……92
4.1.2　认证和授权（访问控制）……92
4.1.3　基于令牌的安全实践……92
4.1.4　安全性的责任……92
4.1.5　编制风格的风险……92
4.1.6　微服务之间的通信……93
4.2　与 OpenID 的 OAuth 2.0 一起使用 JWT……94
4.2.1　OpenID……94
4.2.2　OAuth 2.0……95
4.2.3　JWT……97
4.2.4　示例应用……99
4.3　小结……111
第 5 章　创建高效的数据模型……113
5.1　数据和建模……113
5.2　单体架构中的数据模型……114
5.3　SOA 中的数据模型……115
5.4　微服务架构中的数据模型……116
5.4.1　每个微服务限定一些数据库表……116
5.4.2　每个微服务一个数据库……117
5.4.3　Saga 模式……117
5.4.4　必要时采用混合数据技术……119
5.5　从单体应用向微服务迁移数据模型……120
5.5.1　领域驱动设计……120
5.5.2　数据模型迁移方式……121
5.6　小结……126
第 6 章　测试微服务……127
6.1　微服务中测试的目的……127
6.2　单元测试……128
6.3　集成测试……131
6.4　组件（服务）测试……131
6.5　契约测试……133
6.5.1　Pact……133
6.5.2　Spring Cloud Contract……134
6.6　端到端测试……135
6.7　更进一步……135
6.8　小结……136
第 7 章　部署微服务……137
7.1　持续集成……137
7.2　持续交付……138
7.3　用微服务配置 CI 和 CD 工具……140
7.4　微服务的 Docker 化……147
7.4.1　Docker……148
7.4.2　Docker 引擎……148



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

7.4.3 Docker 镜像 …… 148
7.4.4 Docker 存储 …… 148
7.4.5 应用程序在 Docker 中是如何工作的 …… 149
7.4.6 公共、私有和官方的镜像库 …… 149
7.4.7 Docker 与 VM 的区别 …… 149
7.4.8 在 Linux 中安装 Docker …… 150
7.5 在 Docker 化的微服务中使用开源 CI 工具 …… 154
7.6 小结 …… 156

第 8 章 演进现有系统 …… 157

8.1 从哪里开始 …… 159
8.1.1 架构视角和最佳实践 …… 159
8.1.2 数据库视角和最佳实践 …… 162
8.2 示例应用及其演变过程 …… 163
8.2.1 用户管理服务 …… 164
8.2.2 购物车/订单服务 …… 164
8.2.3 支付服务 …… 164
8.2.4 配送/跟踪服务和通信服务 …… 164
8.2.5 产品推荐服务 …… 165
8.2.6 调度服务 …… 165
8.3 小结 …… 166

第 9 章 微服务的监控和扩展 …… 167

9.1 微服务系统的监控原则 …… 167
9.1.1 如何设置并使用警报 …… 168
9.1.2 从一开始做好监控和发布渠道规划 …… 168
9.1.3 自动扩展和自动发现 …… 168
9.1.4 监控前门流量 …… 169
9.2 监控模式的发展变化 …… 169
9.3 日志记录有助于监控 …… 170
9.4 微服务系统的扩展原则 …… 171
9.4.1 *x* 轴 …… 172
9.4.2 *y* 轴 …… 172
9.4.3 *z* 轴 …… 173
9.5 实施扩展策略前请三思 …… 174
9.6 微服务的监控和扩展工具 …… 175
9.7 小结 …… 180

第 10 章 故障排除 …… 181

10.1 使用微服务时的常见问题 …… 181
10.1.1 性能下降 …… 181
10.1.2 日志记录位置因编程语言而异 …… 183
10.1.3 多组件之间的耦合或依赖问题 …… 184
10.1.4 服务部署数量与日俱增 …… 184
10.1.5 监控多项服务，发现性能下降或其他问题 …… 185
10.1.6 日志与不同组件的关系 …… 185
10.2 常见问题的解决方法 …… 186
10.2.1 解决性能问题的步骤 …… 186
10.2.2 处理不同语言生成的并处于不同位置的日志记录 …… 186
10.2.3 服务之间的依赖关系 …… 187
10.2.4 DevOps 专家积极参与 …… 187
10.2.5 监控 …… 188
10.3 小结 …… 189



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 1 章
微服务架构简介

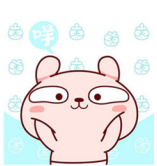
软件架构可以定义为系统设计的一组规则和原则，它定义了软件系统的元素、行为、结构和不同组件之间的关系。

在 20 世纪 80 年代初期，出现了一些大型软件系统，亟需一种统一的模式（也就是后来的架构）来解决设计这些庞大系统所面临的一些常见问题。从那时开始，演化出了今天我们所熟知的“软件架构”的概念。自此之后，很多架构类型被引入到大型软件系统的设计当中。细细数来，软件行业已经见证了**从不共享架构**（shared nothing），到**单体架构**（monolithic），到**客户-服务器架构**（client-server），到**分布式多层架构**（n-tire），再到**面向服务架构**（service-oriented architecture，SOA）等架构风格。**微服务架构**无疑就是这条演化链上的一个新节点。

近年来，微服务这个词的热度在各种软件开发者/架构师社区中呈指数级增长。我们经常听到一些采用了单体架构的组织抱怨发布周期太长、调试烦琐、维护成本高、扩容难等问题。这些问题罄竹难书，以至于即使是少数管理得很好的单体应用也需要花费大量的人力物力来解决这些问题。微服务为解决这些问题提供了一种高效的办法，这也毫无疑问是其日益火热的原因之一。一言以蔽之，微服务架构可以把一个很大、很复杂的问题分解成一系列相对较小的服务，并且每个服务只负责自己分管的那一部分。

微服务架构的基本哲理是：*只做一件事，并把它做到极致。*

微服务的核心是**单一职责原则**（Single Responsibility Principle，SRP）。在微服务架构中，大的业务块会被拆分为一些小的任务，每一个小的任务都依托于一个微服务来完成。在微服务架构的系统中，微服务的数量可多可少，取决于业务需求以及任务被拆分的情况。微服务架构可以给组织带来很多单体架构所没有的好处，但是同时，微服务架构也有自己的一些问题需要解决。我们会在接下来的章节中继续讨论微服务架构的优势和短板。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.1　常规微服务架构

微服务架构的灵感来自面向服务架构（SOA）。微服务架构没有什么黄金法则，如果我们观摩一下现在软件行业中实现的这些不同的微服务架构，就会发现每项应用都有自己的不同风格的微服务架构。关于微服务，并没有完美或标准化的清晰定义。但是总的来说，我们可以通过一些特征和原则归纳出微服务架构。

1.2　微服务架构的特征

任何呈现出下面这 6 个原则和特征的架构都可以划归微服务架构的范畴。

- 系统由两个及以上运行单元或组件组成。这些组件将其自身功能以服务的形式展现出来。每个组件服务于一个业务目标，各个组件之间是松耦合的。组件之间按照预先定义的协议进行交互，协议包括消息队列、HTTP 请求/响应模型等。
- 系统应该是与编程语言无关的，某一个组件可以用 Java 开发，与此同时另外一个组件可以用.NET 开发。单独某一个服务在实现时的技术选型应该不会影响到整个应用的架构。
- 系统的数据库应该是去中心化的。理想情况下，每个微服务应该享有自己的数据库，该数据库仅供该服务自己使用，任何其他组件或者服务都不能提取或者修改该数据库中的数据。
- 系统中的每个组件都应该是内聚、独立且可以自行部署的。任何一个组件在正常工作和部署的时候都不应该依赖其他组件或者资源。为了实现快速上线，每个组件应该有自己的**持续集成/持续部署**（CI/CD）流水线。
- 系统应该有自动化测试。微服务架构最值得称道的特征就是快，在构建、测试到上线的整个周期中，如果没有自动化测试，速度快就是一纸空谈。
- 任何一个组件/服务的故障都应该是隔离的。单个服务的故障不应该拖垮整个应用，也不应该影响其他组件和服务。为了做到这一点，应该有一些故障回滚的机制。意思是说，如果某个服务出现了故障，它应该能够很容易地回滚到之前的一个能正常工作的版本。

下面我们举个简单的例子来具象化我们所说的这些原则。

1.2.1　问题定义

假设我们需要一个能根据注册用户的权限生成打折优惠的在线购物应用。对于铂金



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

用户打八折，黄金用户八五折，白银用户九折，普通访客九五折。

1.2.2 解决方案

在这个架构中，我们有两个微服务。第一个服务用来验证用户权限，这个服务需要接受用户登录时的认证信息，然后将用户的权限等级作为应答返还给消费者。第二个服务接受一个权限等级，然后按照用户的购物车（购物车本身应该是另外一个服务）返回适用的折扣比例。

在图 1-1 所示的架构图中，有两个组件，它们分别有各自对应的数据库。假设这两个组件都以 REST 接口的形式公开其服务。消费者可以在**认证服务 MS1** 中通过登录认证信息得到用户的详细信息，包括用户的权限等级，然后可以在第二个折扣服务 MS2 中，通过权限等级得到折扣百分比。

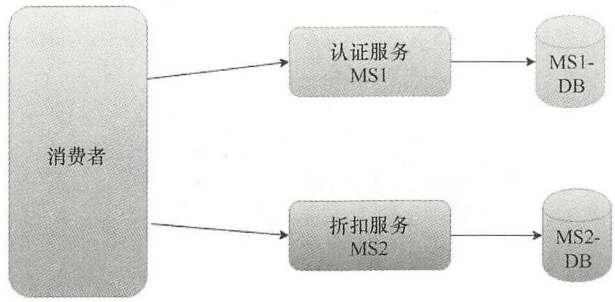


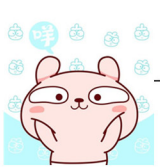
图 1-1

该解决方案贴合了哪些微服务架构的原则

每个服务只为一个业务目的服务（原则 1）。每个服务可以是不同平台的，相互之间无影响（原则 2）。每个微服务都有自己独享的数据库（原则 3）。多个服务之间是相互独立的，也就是说它们不共享各自的数据库，并且可以各自独立部署（假设已经有了相应的 CI/CD 流水线，也就是原则 4）。

现在我们假设在某些极少数的情况下，认证服务 MS1 由于运行时异常或者内存泄漏问题挂掉，此时消费者在访问认证服务时，会得到 404 的 HTTP 状态码响应。404（未找到）会被消费者理解成该用户在数据库中不存在，此时消费者会将该用户理解为普通访客，紧接着就会按照普通访客的方式计算折扣。也就是说，系统始终处于运行状态。认证服务 MS1 故障并没有危害其他正在运行的服务（对应原则 6 故障隔离）。

在传统的单体架构中，如果出现这种问题（内存泄漏），我们就需要重启整个应用，包括其中的所有组件，也就是说整个系统会有一段死机时间。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务架构成功的基础是深入分析问题领域，并将整个应用程序拆分成若干恰到好处的较小的独立组件。拆分做得好，架构就能朝好的方向继续演进。典型的微服务架构的示意图如图 1-2 所示。

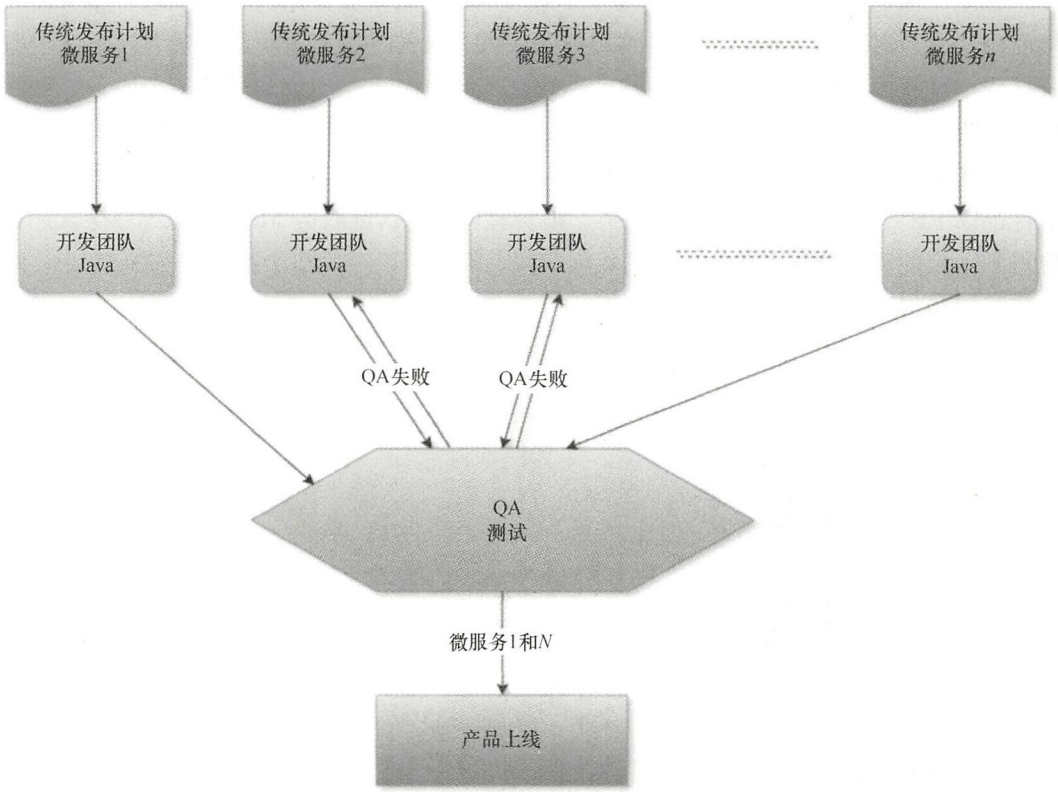
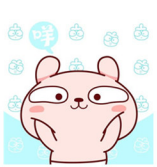


图 1-2

一般而言，每个开发团队只负责一个微服务。根据微服务的大小和复杂程度，团队成员可多可少。每个微服务应该有自己的发布计划，某个微服务的部署过程不会影响到另一个微服务的部署。如果在某个时间点上，有 5 个微服务进展到测试和**质量分析**（QA）阶段，并且只有 2 个通过了质量分析阶段，那么就只有这 2 个微服务可以发布到生产环境，它们的发布不应该受其他微服务的影响也不应该依赖于其他微服务的发布。

1.3　做好微服务架构面临的挑战

微服务架构并不是一个银弹，它并不能解决世界上所有的架构问题。虽然微服务对



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

很多常见的问题给出了一种解决办法，但是微服务也面临着一些挑战。在转向微服务的过程中，我们经常会遇到分解数据库、API 交互、大量的开发和运维工作以及统一团队心态等问题。即便是成功地实现了微服务的主体，也还会面临下面的这样一些挑战。

1.3.1 通过日志调试

对开发人员而言，在微服务架构中调试会变得很艰难。发给应用程序的某一次请求可能会历经多个不同微服务的多次交互才能完成业务处理。每个微服务会产生自己的日志，这种情况下让开发人员来寻找问题的根源简直就是一场噩梦。在分布式日志环境中，调试任何问题都是艰难的。如果我们尝试通过一些合适的工具和脚本把所有服务的日志收集到一个集中的地方，就会需要增加运维团队的工作。

1.3.2 服务监控

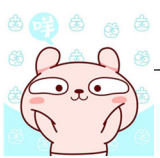
单体架构的应用其实很容易监控。在一个单体架构的应用中，只有很少的几个点需要监控，因为所有功能都在一个服务中。一般的监控点包括数据库服务、磁盘空间、CPU 使用率以及一些第三方工具等。在微服务架构中，监控这一项的开销会呈指数级增长。微服务架构中的每一个微服务都需要一全套类似单体架构的监控。可以想象一下，几百个微服务凑到一块的时候，报警率会是什么情况。一般而言，为了解决和应对不同服务中产生的报警，我们需要使用一些如 Nagios 和 Monit 这样的监控和自愈工具或者脚本。

1.3.3 公共库

在多个服务间共享一个公共库可能不是个好的做法。假设我们有一个 A 服务，它可以生成一个 JSON 格式的用户对象，另外有一个 B 服务会来消费这个 JSON 对象。这个时候常见的做法是，我们将用户类定义在一个 JAR 包中，然后在 A 服务和 B 服务中均添加该 JAR 包的依赖。但是这么做的话会引发两个新的问题：首先，使用公共库的方式会将相关的微服务限定在同样一种编程语言中，也就是在一定程度上丧失了微服务的灵活性；另外一个问题是，这种做法增加了服务之间的耦合程度，这也是与微服务的初衷相悖的。我们觉得比较好的做法是在每个服务中分别定义 `User` 类，当然你可以说我们违反了**避免重复**（Don't Repeat Yourself，DRY）原则。

1.3.4 服务之间的消息传递

在微服务架构中，一个从外部来的请求（来自于前端或者 API 消费者）可能会引起



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

不同服务之间的多次内部通信，极端情况下会引起网络延迟。这样一来，整个应用程序的性能就会变差。解决这种问题的一般思路是，针对不同的场景选择适用的 API 间的通信方式。常见的通信接口有同步的 REST 风格的网络服务接口以及异步的消息队列等。根据不同的业务需要，不同的通信方式可以共存于整个微服务架构中。某些情况下，如果我们需要直接的响应结果，可以采用同步的方式，如 HTTP。而如果某个服务只产生中间结果，还需要将此中间结果传递给下一个服务或者很多个其他服务，那么这个时候就比较适合使用消息队列将事件发布到某个主题或者队列中，常见的产品有 Kafka 和 SNS。

1.3.5　微服务的部署和版本管理

微服务可以用很多不同的方式进行部署。通常微服务是随着容器（如 packer、Docker 等）一起发布的。在使用**亚马逊云服务**（Amazon Web Services，AWS）的时候可以用**亚马逊机器镜像**（Amazon Machine Image，AMI）来发布，但是 AMI 创建和部署的时间会比 Docker 长，Docker 是一个非常好的用来部署微服务的工具。除了部署之外，微服务架构中，还必须采用一些合适的版本规则，否则整个服务群会因为版本问题变得非常混乱。`xx.xx.xx` 就是一种非常典型的三级式版本格式，最右边一级表示一些小型的漏洞修复或者补丁，中间一级代表了一些新功能的升级，最左边一级则通常对应着一个非常大的改动，例如，整个通信的接口模型都发生变化了，等等。

在接下来的章节中我们会详细讨论服务之间通信和部署的相关问题。

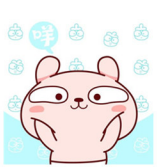
1.4　微服务的未来

基于前面所述的这些优劣处，微服务正变得日益流行起来。随着微服务概念的使用者越来越多，也会出现一些新的问题、新的实现模式以及新的需求。如果我们仔细研究这些新的需求和新的问题，或许就能挖掘出来微服务架构未来的走势和方向。

随着微服务数量的增加，不可避免地会增加运维（DevOps）的工作。例如，在微服务中运维人员需要负责部署多条流水线、集中收集日志、监控服务和服务器参数以及自我修复脚本等。也就是说从运维的角度来看，开发人员除了开发代码来实现业务逻辑之外，还有很多与运维相关的事要做。随着运维工作的日渐繁重，我们目前可以看到微服务的发展趋势主要有下面两类：

（1）无服务架构；

（2）微服务作为 PaaS（平台及服务）服务。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.4.1 无服务架构

无服务架构离我们并不遥远，在不同的云服务平台上已经有了一些相关的成型产品了。例如，亚马逊云 AWS 有一个很酷的 **Lambda**（匿名函数）功能。其主要的工作原理是，将一部分核心代码以 ZIP 文件的形式存储在 S3（AWS 的对象存储服务）环境中，当某个特定事件被触发时，AWS 会按照预定义参数启动一个新的实例（也就是一台 AWS 云虚拟机），然后将 S3 中的代码复制到临时实例中并开始执行这部分代码。代码执行完毕之后，标志着本次任务的结束，此时 AWS 会自动关停该实例。在服务发现的模式下，这一概念变得非常有趣。在这种环境中，服务完全是按照需求来自动生成和结束的。每个服务在启动之后会自行注册，服务停止运行时会自行撤销，而整个过程中仍然是在使用服务注册发现的协议与其他服务进行通信。

上面提到的架构的例子，并没有覆盖到无服务架构的全貌。例如，它并没有对服务的版本化、延迟时间以及一个新的 HTTP 请求到来时如何找到对应的需要启动的服务等问题做出解释。事实上，在这种情形下，实例和服务的启动时间是至关重要的。总而言之，引入了 Lambda 这种类型的组件之后，微服务的未来可以走向无服务架构的方式。无服务架构下，开发人员可以主要专注于开发业务组件，运维工作会少很多。

1.4.2 微服务即 PaaS

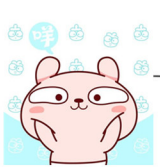
微服务即**平台即服务**（Platform as a service，PaaS）可能是微服务未来发展的另外一条路。微服务可以以**平台即服务**（PaaS）的方式进行交付。这个方案基于现有的一些容器技术，如 packer 或者 Docker，可行性很高。在这个方案中，微服务会与一些成熟的框架一起发布，这些框架一般会提供监控、日志收集、服务注册发现机制、公共库、负载均衡、版本化等相关功能的基础实现。也就是说监控、日志收集等这些烦琐的运维工作在框架里已经预先定义好了，开发人员不用太操心这些东西了，只需要专注于业务代码即可。

随着容器技术的日渐成熟和云组件的发展，未来这两种方式可能会合并到一起来形成一种更优的解决方案。

1.5 与传统架构相比微服务架构的优势

约翰的故事

约翰是一个不错的软件开发工程师，他就职于一家研发股票交易软件的公司。他们



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

公司的股票交易软件有一个功能是给用户推荐一些未来行情看好的股票。这样的推荐主要是基于一些概率分析组件计算得出的。某一天约翰觉得这个概率分析组件可以进行一些微调来提升股票预测的准确度。他觉得只需要在代码里改几行就可以了，看起来午餐之前就可以搞定。他们公司的这个股票交易软件是一个单体架构，里面有大约 40 个组件。结果，约翰花了整整一天时间来复制代码，找出哪些组件会受到牵连以及准备本地环境。第三方工具的依赖、配置不匹配等烦琐的问题都需要花很多时间来识别和解决之后，应用程序才能正常启动。

所以我们可以把约翰在单体环境中遇到的挑战进行一个总结，当然这其实也是所有单体架构中都非常常见的问题。

- 当我们在说单体架构的时候，指的是一个很大的代码库，某些代码库里面分了一些小的子模块，也有些没有分，但是不管分与不分，无一例外的是各个子模块不是各自独立的，当然也不能独立部署。
- 整个代码库是用某一种语言写的，所以所有开发人员都被限制在这个技术栈里了。
- 在这个大的应用程序中，如果负载增加了，只能对整个应用程序进行扩容，哪怕负载的影响只作用在其中的一个模块上。单体应用只能通过给整个应用程序运行多个实例的方式来实现水平扩展。
- 随着代码的增加，在引入新功能的时候，更容易出漏洞。而且由于各个模块可能有些嵌套和依赖，因此随着功能的增加，整个测试周期和部署周期都会无可避免地增加。
- 在单体应用中，大家都不想对整个代码库负责，因为项目实在是太大了。

在另外一种情形下，项目组使用的是微服务架构，此时如果约翰要做个微调又会怎么样呢？首先约翰需要克隆下来微调所在 server 的代码库，需要克隆的代码也不会很多（相比于单体架构的情形）。由于代码本身不多，所以约翰很容易就能找到需要改代码的地方。修改完代码之后，在本地启动该微服务也非常简单。本地验证完之后，约翰开始提交代码和发起代码审查。由于持续集成/持续交付流水线的存在，约翰提交的修改代码会自动合并到相应的分支，比如叫 DevOps，然后自动部署到开发环境。总地算下来，约翰在午餐之前就可以做好新版本上线的所有准备工作了。

虽然上面的这个场景是我们假设的，但其实与很多新加入团队的开发人员遇到的情况非常相似。如果新人来的第一天就能把自己写的代码成功上线，可以明显提振其自信心。而且如果需要依赖的代码库很少的话，新人来的第一天就可以做很多工作。但是在单体应用中，新人光是花在了解整个代码库及其为数众多的组件上的时间就经常长达一周甚至更久。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务架构同样也会给组织带来好处。由于微服务将整个应用程序拆解成了数个小的服务，一般而言每一个服务有自己单独的代码库，更小的代码库意味着更容易理解。微服务也还是符合孤立原则的，因此单独某个服务的部署过程并不会影响其他既有服务。微服务架构下的应用程序可以使用多种多样的编程语言，理论上每一个服务可以用一种不同的编程语言和一种不同的数据库（SQL、MongoDB 和 Cassandra 等）来实现。微服务架构可以容纳非常多的技术栈，因此开发者完全可以按照自己的意愿为他们的微服务选择合适的语言。

1.6 是不是看起来与 SOA 很像

微服务和 SOA 像吗？答案只有一个：绝对像！那么微服务和 SOA 有什么区别吗？答案也是只有一个：绝对有！微服务架构确实受到了 SOA 的一些启发。微服务架构看起来与 SOA 颇为相似，但是下面我们要提到的这些关键不同点让微服务优于 SOA。在 SOA 中，我们也需要把问题拆分为不同的服务，一种典型的通信方式是各个服务之间使用 SOAP 协议。在服务总线上，连接了很多消费者和许多服务。消费者可以通过向**企业消息总线**（Enterprise Service Bus，ESB）发消息的方式来调用任何服务。与在微服务中的情形类似，SOA 里面这些不同的服务也可以分别使用不同的编程语言来实现。SOA 并不强调这些服务的边界上下文，在微服务中虽然不是十分严格，但是一般会按照单一职责原则给出一些服务的边界上下文，也就是服务的边界。SOA 架构中的数据库一般是各个服务所共享的，但是在微服务中每个服务有其独自享有的数据库，其他服务无法直接接触到。SOA 很重要的一个设计原则是“尽可能多地共享”，这也是为什么 SOA 的数据库通常设计成多个服务共享的根本原因。这种共享会增加服务之间的耦合，也是与微服务的理念相悖的。微服务中的原则是“尽可能少地共享”，正是因为这个根本性的不同，导致了两种架构表现出完全不同的设计思路和实现方式。

SOA 中的通信层是 ESB 中间件，会演变成单点故障。如果 ESB 出现了故障，则没有一个服务能正常工作，既无法调用数据，也无法与其他服务进行通信。微服务中各个服务基于孤立原则设计，可以有效规避这种单点故障。微服务典型的通信方式是 REST 风格的 API，这种方式在极大程度上去除了 ESB 这种重量级的作为通信层的中间件。微服务还去除了 SOA 中的服务契约部分，由于在 SOA 中并没有边界上下文的概念，某个服务可以直接修改其他数据模型，为了能这么做就经常需要把数据模型共享出去。这个数据模型可能是个有层级的复杂数据结构。一旦这个共享中的数据结构发生了一点变化，所有牵连到的服务都要修改接口才行，最后还需要把整个应用程序重新部署一遍才能生



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

效。图 1-3 是一个典型的 SOA 应用的示意图。

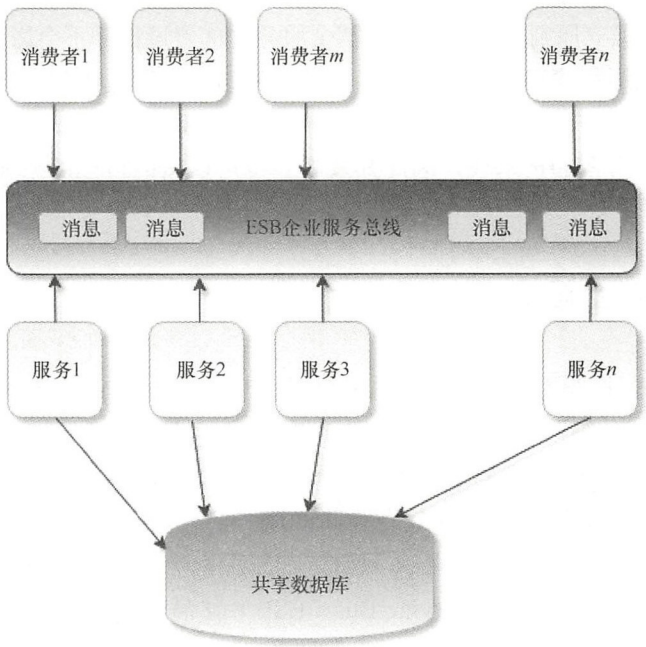
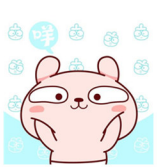


图 1-3

上面提到的这些 SOA 早期的问题现在很多都有办法在架构内部解决和优化，但是不是所有的问题都能解决。SOA 是一个非常广义的概念，我们可以把微服务视为 SOA 的一个子集，一个特例。这个特例通过定义一些更细致的规则实现了更好的面向服务设计。有一句名言是这样描述微服务的：微服务是 SOA 的一种更好的实现。

与单体架构相比，微服务架构的最主要优势在于其上线速度快。不仅初创公司在使用微服务，有很多大公司也在采用，很多使用者选用微服务就是因为微服务可以大大缩短产品**上市时间**（Time to Market，TTM）。市场上的需求始终是动态并随时间变化着的，谁能更快地发布产品谁就能率先抢占市场，因此产品上市时间显得尤为关键。好的产品会根据市场的反应来调整一些业务需求，微服务架构完美的适用于这种频繁修改需求的场景。微服务架构能快速完成产品的开发、测试到上线的整个流程。当今的市场瞬息万变，竞争也十分激烈，用户的反馈声音也愈加重要。产品的基本迭代周期是这样的：研发、测试、上线、收集市场上的用户反馈、针对反馈的后续开发、再次测试、再次上线。如果哪一个产品能比竞争者在更短的时间内交付新功能，或者根据市场反馈进行相应的改进升级，那么该产品无疑会迅速抢占市场，占据主动。在单体架



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

构中做到快速发布上线是很难的，哪怕是一个很小的功能上线也需要牵连到整个应用程序的测试和部署。

1.7 将业务领域划分为微服务组件

要设计健壮的微服务架构，首先标识出业务领域，然后围绕各个领域定义该领域应具有的功能，最后再围绕这些功能和领域进行微服务的划分和开发。业务领域的划分可以使用**领域驱动设计**（Domain-Driven Design，DDD）的方式来进行。领域驱动设计是一种解决复杂需求的软件开发方式，使用领域驱动设计可以将具体实现和演化模型很好地联系起来。

DDD 这个词是 2004 年引入的，一并引入的还有这样一些关键词：建模、边界上下文、实体、代码库和通用语言等。DDD 和微服务的理念很契合，因此 DDD 也随着微服务的流行而逐渐广为人知。DDD 的理念是尝试理解问题的领域，然后围绕领域进行建模。DDD 建议使用者基于真实的用户案例进行建模。建模过程主要是围绕领域，定义一组概念、功能和边界上下文。DDD 有这样一些基本概念。

- **通用语言**：指的是所有的利益相关者（业务部门、运营人员、开发人员）应该使用相同的语言（术语）来描述问题，这种语言不应该太偏向技术方面，而应该更加贴合业务方的模型。该通用语言应该和领域的边界上下文有关。
- **边界上下文**：边界上下文的主要目的是用来定义模型的范围和清晰的上下文边界，然后尽最大可能保持模型统一。最终定义出的模型应该是各方面达成一致的，而且是限定在清晰定义了的边界内。每一个边界上下文都应该是与其他的上下文相互独立的，边界上下文定义了不同领域所分担的职责。某些业务可以设计多个领域，如货运、库存、销售、市场等。每个领域有其自己的功能。每个实体或业务组件都应该只存在于其自己的边界上下文中。DDD 的概念可以帮助我们识别出来这些领域和他们的分离点。整个 DDD 的概念都符合微服务的原则。围绕模型进行思考，按照边界上下文对模型进行封装，这些上下文就清楚地定义了模型的边界。

微服务是彼此隔离的，因此对于微服务的职责和边界进行合理的划分显得非常有必要。如果采用 DDD，我们可以通过边界上下文将一个大的问题分解为一些小的问题，进而可以让我们更好地理解和定义微服务。如果职责没有划分清楚，就会在微服务之间产生职责泄露，然后系统会在这些边界不清楚的地方变成“大泥球”。DDD 可以帮助我们理解和定义边界上下文。关于领域驱动设计有一本不错的书可以推荐大家阅读，叫 *Implementing*



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Domain-Driven Design[①]，作者是 Vaughn Vernon。通过这本书读者可以大大加深对领域驱动设计的理解。

围绕业务功能组织微服务组件

微服务可以给组织的交付速度赋能。使用微服务的组织可以更快地响应市场变化，可以极大提高生产力、交付能力以及项目交付速度。然而选择微服务的组织中，一个很大的挑战是组织结构同样需要围绕微服务架构进行一些调整。可以这么说，微服务并不适用于每个组织。实现微服务也在一定程度上会动摇一些组织结构。

业务功能可以定义为用于特定业务目的的一个独立业务单元。组织中的各个层级和领域可能都会定义业务功能。从管理层到运营部门再到销售和技术人员，不同部门可能会有一些不同的业务功能的定义。因此我们要识别适合的业务功能的粒度，然后围绕这些业务功能安排微服务。

业务功能和微服务从定义上看像是同一个东西，但是它们是不同的。例如，搞一个促销活动可能是市场部门的一个业务功能，围绕这个功能我们需要一个电子邮件发送服务、一个促销链接生成服务和一个数据收集服务等。这是一个两步的过程：

（1）识别正确的业务功能；

（2）围绕该业务功能定义微服务。

这两步缺一不可，而且要划分好粒度，只有这么做才能设计出好的微服务架构。

另外一个重要的因素是沟通交流。沟通在任何形式的业务中都是必不可少的。团队与团队之间需要沟通，组织与组织之间也需要沟通，收集终端用户的持续反馈同样也需要沟通，组织内部各个不同的管理层之间需要沟通，甚至组织内部软件系统的不同组件之间如何协同工作都需要沟通。有这样一条定律将软件系统的沟通方式和组织结构的沟通方式联系了起来，它就是康威定律：*设计系统的组织，其产生的设计的结构等同于组织之内、组织之间的沟通结构。*

康威定律对于很多初次接触的人来说有些难以置信。一些知名院校（如麻省理工学院和哈佛大学）认真研究过这个问题。他们在这个定律上做了一些研究，他们比较了不同组织结构中研发出来的代码，最后发现真的是符合康威定律的。他们采集到的代码中，90%都是完美符合康威定律的。为了理解这一点，我们以单体架构举一个例子：如果你有一个庞大的开发团队，所有的开发成员都在同一个地方协同办公，那么你最后做出来的肯定就是一个单体应用；如果都是一些小的团队，团队内大家在一起协同开发，但是团队与团队之间是不需要待在一起的，那么最后做出来的东西肯定是分布

① 这本书中文版书名《实现领域驱动设计》，译者是张逸和滕云。——译者注



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

式的或者多模块的，就像很多开源项目一样。开源项目一般都是分布式的和多模块架构的，因为它们通常都是由很多小团队（某些团队小到只有 1 个人）而不是一个很多人的大团队完成的，这些小团队之间不是一直在协作沟通的。康威定律虽然没有提到任何与微服务相关的东西，但是康威定律告诉了我们团队结构或组织结构将如何影响软件应用的设计。

根据康威定律，软件应用的架构最后会是组织关系的一个副本，组织结构很可能会影响应用架构，团队所处的工作地点和团队沟通方式也会影响应用架构，软件应用的架构最终会与组织结构一致。

反向亦是如此，例如一个组织里面有 5 个开发团队，每个团队有 4 个开发人员，某一天我们遇到一个棘手的问题，不好拆分到某一个具体的服务里面交由某一个团队单独完成。这个时候可能不得不把两个团队重新组合成一个团队。这就是一个软件架构反向改变组织架构的例子。

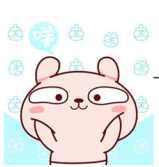
事实情况是，每个公司都有其自身的规范、业务功能和目标以及团队结构和沟通方式。在实施微服务的过程中，或多或少会改变这些既有的组织结构。所以如果想做好微服务的话，首先需要做的功课是了解清楚组织里面当前的组织结构和沟通方式。

1.8 到底要不要使用微服务

微服务许诺过一些单体架构中难以实现的特性和功能，如可扩展性强、高可用、组件之间低耦合高内聚等。但是在落实微服务的时候不可避免地会遇到一些挑战。迁移成微服务的决定并不容易，需要思虑周全。而且这个概念还比较新，目前成熟的经验不多。随着越来越多拥趸的加入，我们希望在未来可以提炼出来一些成熟的模式。在一个组织准备转型成微服务架构时，需要考虑很多因素，如我们下面列出的这些。

1.8.1 组织认同度

确保所有即将参与到微服务转型中的人都接受过相应的培训。这个培训不只是纯技术技能，而是整个微服务的相关概念。大多数情况下，团队应该采用全栈开发的理念，也就是开发人员都需要知道一些 DevOps 知识。微服务架构中的开发人员对于其分管的微服务要有完全的支配权。让每一个开发人员意识到他们的职责不再按照前端开发、后端开发、运维等划分是很有必要的。团队要时刻准备好自己搞定所有的问题。如果有专门的 DevOps 人员当然好，但是每个团队还是要储备好 DevOps 相关能力做到自给自足。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.8.2　体验 DevOps

确保组织中有一个成熟的运维团队和一天之内处理多个上线安排的流程。在单体架构向微服务架构转型的时候，每个服务需要单独进行监控，这对于基础设施和 DevOps 团队来说是一项额外的职责。如果想做到快速独立发布，就需要格外注意 DevOps 和监控系统，以确保上线无误或者出现问题之后能够及时回滚。

1.8.3　分析现有数据库模型

数据库的有效拆解是个大问题。某些时候现有的数据库设计得很难进行拆解，或者拆解时需要大幅调整一些数据库表结构的设计。你的组织中的数据库是什么情况呢？是否足够容易拆解成数个物理上和逻辑上有区别的小数据库呢？因为微服务中的数据是分散在多个不同的数据库中的，也就意味着数据库表之间没有任何的物理连接，这将导致大量代码的改动。

1.8.4　自动化和 CI/CD

微服务需要一些自动化测试工具和 CI/CD 工具如 Jenkins 和 Team City 等来支持频繁交付和上线。每一个服务应该有其自己单独的部署流水线和配置管理。这又会增加一些 DevOps 的开销。TDD（测试驱动开发）在这个方面非常重要。微服务需要单元测试和集成测试等自动化测试手段来保证持续交付。

1.8.5　集成

由于应用的不同部分运行在不同的地方（各个微服务中），因此需要做大量的集成工作来保证最终输出的正确性。在向微服务转型时，集成过程中的沟通和集成点的维护工作也是重要的开销来源。

1.8.6　安全

由于微服务中有大量的组件需要通过 API 或者消息队列相互通信，因此在通信过程中一定要考虑到安全性。基于凭证和基于令牌的微服务提供了很多解决安全问题的方案，但是对于开发和基础设施团队来说都会增加一些开销。

任何一个准备转型为微服务架构的组织都需要综合考虑上述因素。成功的实施微服务可以给组织带来极大的可扩展性、高可用性，而且可以无宕机随时部署上线。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.8.7 成功迁移的例子

Netflix 公司在 2009 年成功完成了跨越性的由传统单体应用向微服务架构的转型。而且他们把自己在转型过程中得到的经验分享给了全世界。时至今日，Netflix 有超过 800 个正在运行的微服务，每天会发生超过 20 亿个请求，这 20 亿个请求会在整个系统内部的微服务之间产生超过 200 亿个内部 API 的访问。

在 Netflix 开始从单体应用向微服务应用转型之时，微服务这个词还没有出现。Netflix 之前的单体应用非常庞大，他们最开始是从视频编码代码库迁移过去的，这是一个非面向客户的应用。在迁移成微服务架构的过程中，有一个最佳实践是：在单体架构向微服务迁移的时候，始终从非面向客户的组件开始。这些非面向客户的组件即使迁移失败了，也不会影响产品的主要业务流程。有了最开始的成功经验之后，他们才开始着手迁移那些关键的面向客户的组件。到 2011 年年底，他们的产品已经整个迁移到了 AWS 上以众多微服务的方式运行。Netflix 还开源了很多曾经帮助他们成功完成迁移的重要工具。

1.9 示例项目（信用风险评估引擎）

在本书中，我们会以一个信用风险引擎的项目作为示例。我们不会花过多篇幅讲解信用风险评估的算法。我们会以 2～3 个评估风险是否可控的规则作为例子。这个应用程序的重点不在于风险评估是如何完成的，而在于我们应该如何设计微服务，如何保证安全性，以及各个微服务之间如何通信，等等。在我们的例子中，会使用 Spring Boot 作为我们的开发环境，部署微服务会用到 Docker。

1.10 Spring

Spring 是一个非常令人称奇的框架。不管是曾经用过 Spring 还是现在正在用 Spring 的 Java 程序员都会领略到 Spring 的便捷。如果你用 Java 但是还没有了解过 Spring，那么真的应该停下来反思一下了。Spring 是一个基于 POJO（简单 Java 对象）的轻量级框架。Spring 最大的特点是其依赖注入方式，在 Spring 中创建 bean 和对象的职责交给了 Spring 框架来统一管理、完成和维护。Spring 鼓励使用模块化的方式进行编程。AOP（面向切面编程）和事务管理是很多项目都需要的，这些在 Spring 框架提供的库中都已经内置好了。

在架构层面，Spring 由一个核心容器和很多个其他的模块来搭配满足各种各样的需



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

求。Spring 核心容器包含了以下模块：

- Spring 代码模块；
- Spring bean 模块；
- Spring 上下文；
- **Spring Expression Language（SpEL）**。

基于上面的这些核心模块，Spring 还提供了很多其他的模块，用于构建健壮的基于 Java 的应用程序，如数据库访问模块、事务管理模块、安全模块、Spring 集成模块、批处理模块、消息模块、社交插件、Spring Cloud 等。有很多组件可以让开发变得更加简单高效。单说 Spring 框架的好处，就可以写上一整本书，但是这并不是本书的目的。概括来讲，Spring 的这些组件对于帮助 Java 程序员高效开发大有帮助。但是 Spring 有一个小的瑕疵就是其较为复杂的配置系统。平均一个应用程序可能需要用到超过 4 个 Spring 模块（如核心容器、Spring MVC 事务管理、数据库访问等必不可少的模块），随着项目的发展，用到的模块数会更多，此时各个不同版本的组件之间的配置管理和兼容性就变得十分麻烦了。为了帮助开发人员解决这个问题，Spring Boot 就面世了。

1.11　Spring Boot

Spring Boot 比 Spring 更适合快速启动和易于开发，它省略了很多累赘的 Spring 配置。Spring Boot 不是另外一个独立的开发框架，它只是在 Spring 的基础上做了一些组件的封装，以降低开发的工作量。Spring Boot 为很多样板代码、XML 配置和注解节省了开发的工作量。Spring Boot 可以很容易地与其他 Spring 模块集成，而且还为 Web 应用提供了一个内置的 HTTP Web 服务器。Spring Boot 还有命令行界面。Spring Boot 可以使用多种构建工具，如 Maven 和 Gradle 等。

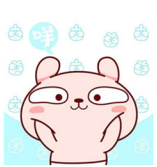
使用 Spring Boot，一切都变得很简单

下面我们举个例子来证实我们前面对 Spring Boot 的赞美。这里我们开发一个只包含一个控制器类的小应用，这个控制器类可以根据用户在 URL 中发过来的名字信息，向用户提供问候功能。

需要为这个例子准备下面这些环境：

- Java 8；
- STS（Spring Tools Suite）；
- Maven。

在 STS 中创建一个新的 Maven 工程，命名为 `FirstBoot`，然后将项目中的 `pom.xml`



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

文件修改成下面这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>
<groupId>com.sample</groupId>
<artifactId>firstboot</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.0.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<!-- more dependency if required , like json converter -->
</project>
```

这里使用的 spring-boot-starter-web 依赖中封装了大量的其他依赖，包括 Tomcat 的 Web 服务器。

在/src/main/java/com/sample/firstboot/controller 目录下，创建一个名为 SampleController.java 的文件，这就是我们定义的控制器了，它将用来处理 Tomcat 接收到的 HTTP 请求。该文件的内容如下：

```
package com.sample.firstboot.controller;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
public class SampleController {
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    @RequestMapping("/greeting/{userName}")
    String home(@PathVariable("userName")  String userName) {
        return "Welcome, "+userName +"!";
    }
}
```

这就是一个简单的控制器类。这里的@RequestMapping 注解用来标识方法可以处理的 URL，这里这个方法能够处理的 URL 是/greeting/{username}。在接收到来自这个 URL 的请求时，控制器会返回 Welcome 加一个用户名。默认情况下处理的是 HTTP GET 方法。这是一个简单直接的控制器的例子。

下一步，创建一个名为 SampleApplication.java 的文件，内容如下：

```
packagecom.sample.firstboot.controller;

importorg.springframework.boot.SpringApplication;
importorg.springframework.boot.autoconfigure.SpringBootApplication;
importorg.springframework.context.ApplicationContext;

@SpringBootApplication
public class SampleApplication {

    public static void main(String[] args) {
        ApplicationContextctx = SpringApplication.run(SampleApplication.class, args);
        System.out.println("Application Ready to Start. Hit the browser.");

    }

}
```

可以看到，这里使用的是一个@SpringBootApplication 注解，而不是一大堆 Spring 框架下的注解组合，这里我们无须再添加诸如@Configuration、@Component、@EnableAutoConfiguration 之类的 Spring 注解。

接下来我们要运行这些代码，打开命令行窗口，然后进入应用程序所在的目录，然后敲入 mvnspring-boot:run 命令。

这一命令会查找到包含@SpringBootApplication 注解的文件并运行这个应用程序。成功执行这个命令之后会有下面这样的日志输出在命令行中：

```
Started SampleApplication in 2.174 seconds
Application Ready to Start. Hit the browser.
```

此时此刻，这个 Spring Boot 应用程序已经可以进行测试了。要测试这个应用程序，需要打开浏览器，然后敲入 http://localhost:8080/greeting/john 这个 URL，



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这时应该能看到图 1-4 所示这样的页面。

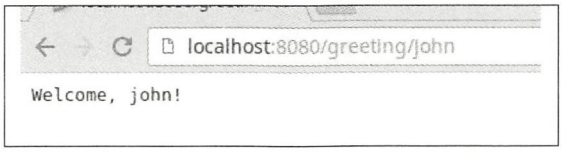


图 1-4

这里的 localhost 地址指向的是用户的计算机，8080 是 Spring Boot 中 Tomcat 启动的默认端口。/greeting/john 是我们在控制器中定义的 URL 的一部分。这里 john 这个名字可以任意替换，如 xyz 等。

由此可见，使用 Spring Boot 开发应用程序是非常简单的。使用 Spring Boot 之后，开发人员无须再去关心复杂的配置和依赖，只用把精力集中在业务逻辑开发上，从而可以极大提升开发效率。

1.12 小结

在本章中，我们学习了微服务的概念，了解了微服务架构的一些特征。读者应该清楚使用微服务架构的好处和一些需要权衡的地方。在下一章中，我们会以一个示例项目，也就是前面提到的风险评估系统为例，介绍微服务架构的具体内容。这个示例系统会贯穿在本书的每一章中。示例风险评估系统会使用 Spring Boot 作为开发框架，使用 Docker 来进行部署。这些技术在本章中都有所提及。接下来我们该干什么了呢？下一章我们会围绕这个风险评估系统，将阐述怎么定义微服务和服务的发现。而且，我们还会使用 Spring Boot 开发我们的第一个微服务——用户注册服务。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 2 章
定义微服务组件

通过上一章的介绍，我们已经了解了微服务架构的一些概况和使用微服务时应该注意的问题。微服务不是一颗万能的银弹，因此微服务不能直接在每一个组织中实施。组织在选择使用微服务架构之前需要很多知识储备。当组织储备好这些微服务的相关知识，并决定迈向微服务之后，接下来的一个重要问题就是：如何定义微服务？

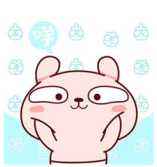
2.1 微服务的定义

微服务并没有一个统一公认的定义。维基百科是这么定义微服务的：

> 微服务是面向服务型结构（SOA）的一种更具体、更现代化的理解，SOA 主要用于构建分布式软件系统。和 SOA 类似的是，微服务架构（MSA）中的服务是一些通过网络互相通信来实现目标的进程。

这个定义里并没有明确说明如何定义微服务。对于微服务的定义，有很多人发表过许多不同看法。大部分看法都可以找到共同点，如单一职责、松耦合、高内聚、服务于业务目标、轻量级通信、很少的代码行数（10～100 行）。当然代码行数这一点有待商榷，因为不同的编程语言有不同的语法和开发库，所以代码行数一般没有统一标准。

亚马逊提出了这样一个词——双比萨团队。其含义是一个微服务团队规模最好是大家一起刚好能吃完两个比萨，也就是说，最多 12 个人的团队规模是理想的，这其中既包含了测试人员也包含了开发人员，如果是用敏捷的方式开发，可能还需要一个产品负责人（product owner）。因此在定义微服务的时候，一定要记住一点，如果某些特征总是一起发生变化的，那么它们就应该待在一起，也就是它们需要放在一个微服务里。微服务的另外一种定义是：一种为了达成某些业务目的，可以独立部署、小型、模块化的、服务之间通过定义好的轻量级协议进行通信的软件应用。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

每个微服务应该有多大的问题，其实是留给整个行业的。随着越来越多的人带着各种不同风格的微服务实践加入到这个行业中来，可能会对形成微服务设计模式有一定的指导作用。就当下而言，我所理解的微服务设计模式可以表述为：独立部署，单独回滚，服务于某个单一业务，可以用一句简短的不带连词的话描述清楚。

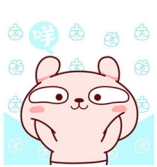
下面几节会集中讨论微服务中的两个重要的环节——**服务发现**和 **API 网关**。在定义微服务的时候，这两方面会起到很重要的作用。

2.2　服务发现及其用途

微服务需要互相通信，为了能相互通信，首先需要知道对方的位置。一种可行的办法是，我们将微服务的 URL 路径绑定到某个固定的 IP 上，但是问题是如果某个微服务的 IP 发生了变化，所有依赖于这个微服务的其他服务都不得不相应地修改 IP 值，然后做一次重新部署。一个可能的解决办法是，我们从代码中把 IP 提取出来，放在一个外部的配置中，下一节我们会专门讲微服务的外部化。如果只有几个微服务，这么做确实可以解决问题，但是随着系统的逐渐演化，微服务的数量也会日渐增多，随着服务数量的增多，环境会变得越来越动态。随着负载的变化，同一时间启动和停止的微服务也会越来越多。这样一来会让识别微服务或者微服务的服务器位置的过程变得越来越困难。而且微服务的监控也是一个难题。在这种情形下，识别某个特殊微服务的位置就不那么简单了，也就是说这种原始的记账方式已经无法满足系统需要了。我们需要一种能快速定位到系统中某个服务的位置的机制，这种机制可以保证开发人员能够以任意顺序随意启停服务，而且当某些微服务发生变化的时候可以很方便地重新配置。为了能做到这些，这种机制或者外部服务应该有足够的能力可以提供微系统中各个微服务启动和停业的动态信息。现在有很多种办法可以解决服务定位的问题，读者可以在适当学习之后根据自己的系统架构选择适合自己的方法。

2.2.1　DNS

首先我们可以考虑用 DNS 来实现。DNS 是域名和 IP 地址的映射表。一个服务可以对应多个 IP 地址。那么如果新加入了某个服务的时候应该怎么处理呢？我们需要给 DNS 添加一个新的映射。此外，如果我们要修改某个服务的 IP 地址，则 DNS 中的修改不会立即生效，而是需要等待一个 TTL（生存时间周期）才能真正生效。DNS 的好处是简单直接，易于实现。但是在一个频繁变化的动态微服务环境中，DNS 的灵活性堪忧。在商业项目中，经常需要在负载高峰期启动很多个服务的副本实例，此时如果任何一个服务



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

需要与这些新加入的实例进行通信，使用 DNS 是很难做到的，因为按需添加和移除实例时，在 DNS 中修改映射表不会立即生效。

2.2.2 服务发现的请求

看过了 DNS 在这种动态环境中的局限性之后可以发现，我们其实需要一个专门解决动态问题的服务才行。这个服务应该有下面这些功能：

- 可以对其自身进行注册和撤销注册；
- 可以将其自身的改动通知给其他所有服务；
- 可以通过该服务定位到某个特殊服务的具体实例。

这些功能是通过发现服务来完成的。发现服务是一种记录了系统中所有可用服务示例的数据库。每一个启动和停止的微服务都应该在这里进行注册和撤销注册。任何一个微服务都可以在这里得到其他微服务实例的元信息，包括主机 IP、端口、服务提供方等其他细节信息。发现服务可以动态更新，因为每一个服务都是在这里进行注册的，所以该发现服务需要给调用方提供即时、准确的被调用服务的实例信息。被调用的服务和发起调用的服务所在的机器、私有网络 VPC、地区等有可能相同也有可能不同，不管是何种情况，发起调用的服务都不用关心，只需要通过发现服务来查找被调用服务即可。此外，为了保证可用性，在发生变化时每个微服务有责任将其自身修改通知给发现服务，如服务名、服务状态、主从状态等。目前的市场上已经有了一些可用的做发现服务的产品了。例如，以下厂商的产品：

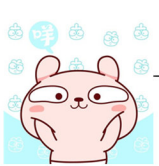
- CoreOS 的 Etcd；
- Netflix 的 Eureka；
- HashiCorp 的 Consul；
- Apache ZooKeeper；
- SkyDNS；
- Airbnb 的 SmartStack。

这些产品不是按推荐顺序列出的，但是我们鼓励读者优先了解 Consul、ZooKeeper 和 Eureka。这 3 个在 Spring 中支持得较好，熟悉 Spring 的开发者应该很容易上手。

1. 服务注册和撤销注册

为了让发现服务这种模式能正常工作，最重要的是要保证发现服务中的服务是随时更新的。任何一个新出现的、销毁的或者进入维护状态的服务的任何一点改动事件都应该被发现服务记录下来。关于服务注册，有下面两种常用的机制：

- 自注册；



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- 第三方注册。

在自注册机制中，每一个新加入到系统中的微服务应该在启动之时将其自身注册到发现服务中去。当服务停止运行的时候，也需要通知发现服务。在收到这些通知之后，发现服务就可以相应地将服务标记为可用或不可用状态。自注册模式看起来较为简洁，但是它违背了单一职责原则。此种模式下，微服务多了一项职责：将自身状态变化事件通知给发现服务。

图 2-1 展示了一种自注册的方法。在这个图中，服务 B 在通过发请求给发现服务来查找服务 A 的实例地址。当服务 A 启动时，它会在发现服务上注册自己，自此之后，发现服务就可以将服务 A 的状态和元数据共享给服务 B 了。

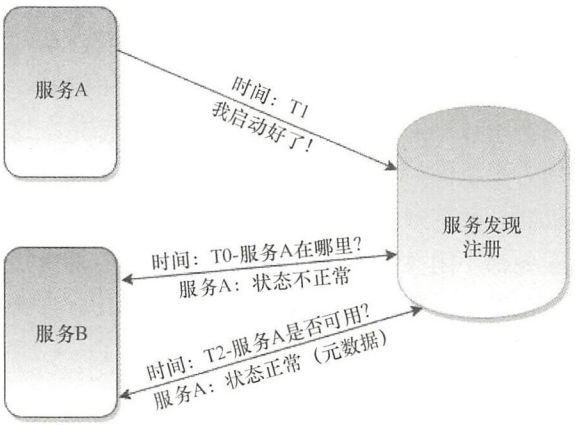


图 2-1

在第三方注册模式下，服务发现会与一个单独的进程一起负责监控所有运行中微服务的状态。这个新进程使用不同的机制来检查微服务的健康状况，如轮询、心跳检测或者访问微服务健康检查 URL 等，并且可以通过收集到的信息来更新各个服务的状态。当然还应该有一些详细的判断微服务是否出现故障的机制。

2. 发现其他服务

除了给其他服务提供注册功能之外，这个特殊的发现服务还有一个功能是提供系统中所有其他服务的元数据。在任何一个时间点上，只要某个服务想要与其他服务进行通信，它就应该可以通过这个发现服务来定位到具体微服务的地址。服务发现也有两种不同的模式：

- 客户端发现；
- 服务端发现。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在客户端发现模式中，每个微服务直接与发现服务进行通信。在访问到具体的微服务之前，调用服务先从发现服务上得到所有其他服务及各自实例的网络地址，然后通过一些负载均衡算法从这些地址中选择出合适的访问服务的实例。Netflix Eureka 采用的就是这种模式，在下一节中我们将举一个在 Spring 中使用 Eureka 的例子。Ribbon 是另外的一个与 Eureka 配合的工具，它可以高效地帮助调用服务选择实例来做负载均衡。

客户端发现模式简单直接，成熟的工具也很多。与自注册模式类似的是，它也破坏了微服务的单一职责原则。在这种模式下，每个微服务还需要负责查找其他服务的网络位置，相应地需要包含进来一部分查找算法来做负载均衡。但是由于这种模式易用，所以还是有很多组织使用这种模式的。

在服务端发现模式中，服务首先将请求发给一个中间件实体，一般是负载均衡器或者网关，然后这个网关与发现服务进行通信来找到所需服务，再通过某些逻辑找到具体的服务实例。API 网关就有这种逻辑可以管理负载，从而选择到某个服务的最佳实例。服务端发现模式符合微服务的单一职责原则。在这种模式下，微服务可以专注于其自身的业务逻辑，我们会在下一节集中讨论 API 网关模式。

不管你选择的是哪一种模式，有了服务提供的信息，服务的功能就更多了。随着服务状态变得健康或者不健康，服务发现还可以提供实例的主从信息。根据操作类型的不同（读或者写），服务可以选择最合适的实例来完成。发现服务还可以提供服务的版本信息。例如，X 服务有两个不同的运行版本，Y 服务想调用 X 服务，此时发现服务可以同时提供 X 服务两个版本的信息，然后网关或者 Y 服务自己可以从这两个版本中选择最合适的版本进行通信。也可以设计成 Y 服务在调用时直接指定所需的 X 服务的版本，然后发现服务直接返回所需版本的 X 服务地址。这个功能可以解决在同一环境中的服务同时运行多个版本的问题。

发现服务还有助于减少调用时的网络延时。在之前的例子中可以看到，在查询 X 服务位置的时候，发现服务可以提供距离 Y 服务最近的实例地址。Y 需要的某个版本的 X 服务可能出现在同一个机器、VPC 或者区域中，只需要稍加上一些判断逻辑，发现服务可以做到智能选择最佳实例来进行通信。这样最终可以在一定程度上减少网络延时。

发现服务可以找到所需服务的健康实例，通过某些智能判断逻辑，可以找到物理上更接近请求服务的实例。发现服务还可以承担某些负载均衡的工作，优先选择负载最小的服务。所以服务注册发现的模式在微服务这种动态的环境中，有更多的灵活性，也可以将失败的风险降到最低。在下一节中，我们会通过一个小的例子，来讲解发现服务是如何工作的。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.2.3　服务发现模式示例

这种模式有很多可用的实现，其中一些已经在前面的章节中提到过了，在这个例子里，我们会选用 Netflix 的 Eureka。Spring Boot（1.4.0）中内置支持了 Netflix Eureka。通过一些简单的注解，我们可以很方便地创建注册服务和注册客户端。在这个例子里，我们会创建一个注册服务，以及一个在启动时进行自注册的客户端。我们首先会使用 Spring 初始化器或者 Maven 原型来创建一个 Eureka 服务的样例代码。就这个特殊服务而言，我们需要引入一些依赖来定义 spring-cloud-starter-eureka-server。Maven 的 Pom 文件中 Eureka 相应的依赖是下面这样的：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Spring 也有 zookeeper 和 consul 的实现，这很容易作为 spring-cloud-starter-consul-discovery 依赖和 spring-cloud-starter-zookeeper-discovery 依赖嵌入。

在主应用类中，用户只需要简单地添加一个@EnableEurekaServer 注解，代码如下：

```
package com.practicalMicroservcies;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

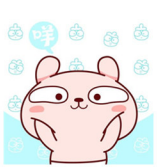
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServerDemoApplication.class, args);
    }
}
```

这里的@EnableEurekaServer 注解启用了 Eureka 服务发现服务器，在 application.properties 中（位置为 src/main/resources/application.properties）添加下面的属性：

```
server.port=8761
```

然后通过 mvn spring-boot:run 运行应用程序，此时用户会看到这样一条警示信息：



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
WARN 24432 --- [nfoReplicator-0] c.n.discovery.InstanceInfoReplicator:
There was a problem with the instance info replicator
```

这是因为在默认情况下 Eureka 服务器中使用的是复制模式，它会查找另外一个复制节点，但是却没有找到，所以报了一条警示信息。而且这个服务还会尝试注册它自己，因此我们需要添加下面这两个属性来解决这个问题：

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

在添加完这些属性之后，服务器就应该启动并在 8761 端口上运行了。此时，如果在浏览器的地址栏中敲入 http://localhost:8761，可以看到图 2-2 所示的仪表盘页面。

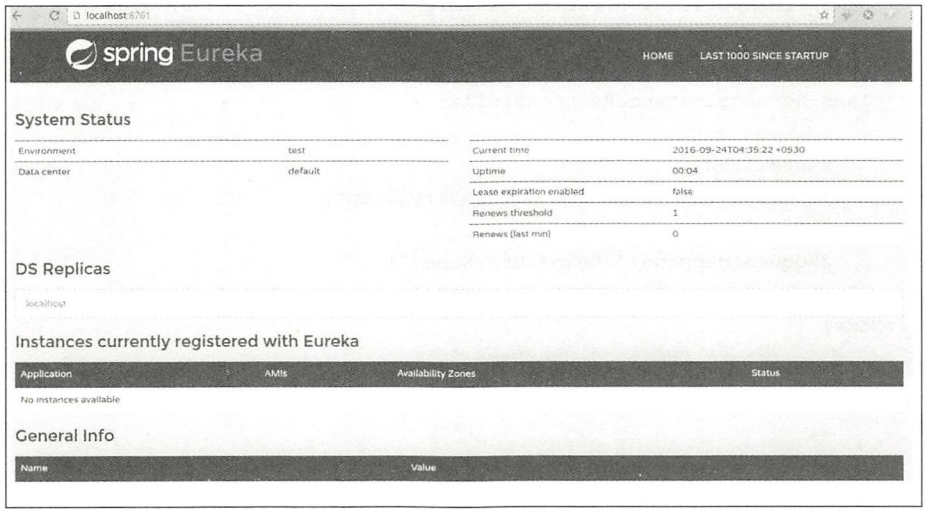


图 2-2

正如所见，在 Instances currently registered with Eureka 标题下没有任何条目，这是因为现在这个服务器还没有注册过任何服务实例。除此之外，在这个页面上还可以在 **General Info** 下面看到服务器的一些基本信息，如 CPU、内存、IP 地址等。于是，通过寥寥数行代码的注解和属性配置，我们的服务注册服务器就已经启动了。

下面我们来创建一个 Eureka 客户端，我们创建一个名为 EurekaClientApplication 的应用/工程，然后生成下面这些 Spring Boot 的模板代码：

```
package com.practicalMicroservcies;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@EnableDiscoveryClient
@SpringBootApplication
public class EurekaClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplication.class, args);
    }
    @RestController
    class ServiceInstanceRestController {

        @Autowired
        private DiscoveryClient discoveryClient;

        @RequestMapping("/Demo/{UserName}")
        public String serviceInstancesByApplicationName( @PathVariable String
UserName) {
            return "Hello "+UserName;
        }
    }
}
```

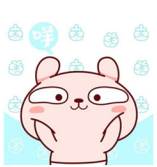
为了测试这个服务，可以测试访问 http://localhost:8080/Demo/Alice 这个 URL，服务会返回 Hello Alice 作为响应。

用了@EnableDiscoveryClient 注解之后，客户端只要一启动，就会开始查找服务注册并对其自身进行注册。注册的名字来自 application.properties 文件中 spring.application.name 这个键对应的值。这个例子中使用的是 PracticalMicroservice-Demo 这个名字。在客户端日志中，应该能看到下面这样的日志行：

```
c.n.e.registry.AbstractInstanceRegistry  : Registered instance
PRACTICALMICROSERVICE-DEMO/192.168.1.4:PracticalMicroService-Demo with
status UP (replication=false)
```

我们可以在服务器的控制台中看到同样的名字，如图 2-3 所示。可以通过下面的代码获得任何一个服务的实例列表：

```
this.discoveryClient.getInstances(serviceName);
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

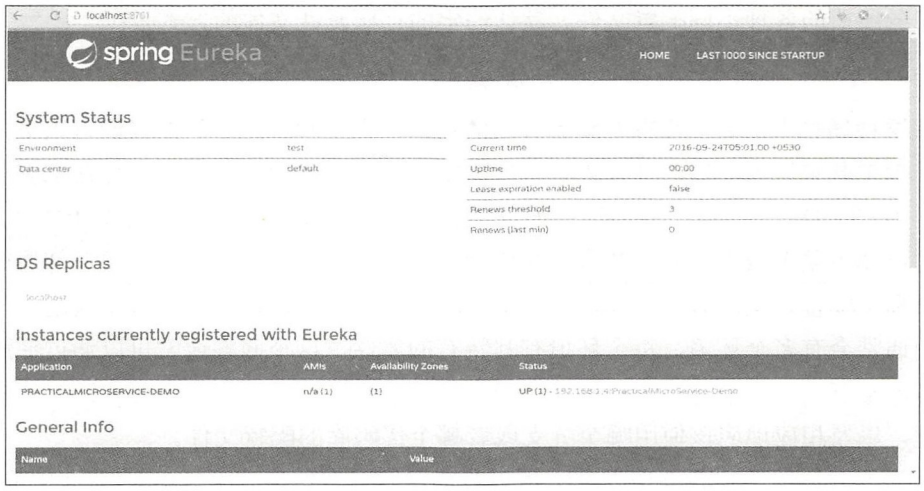


图 2-3

在默认情况下，这种方式实现的服务客户端，会在本机查找服务注册。当然我们可以通过在属性文件中增加服务 URL 来修改配置属性。服务注册过程可能比较缓慢，因为服务器需要给新加入的服务创建一个实例，然后向实例中填充来自服务客户端的各种元数据。即使在注册之后，客户端与服务器端每间隔 30 秒也会有一个心跳检测，当然这个间隔时间是可配置的。

Spring 有很多支持微服务架构和微服务模式的功能。在 Spring Cloud 中，加入了一大批处理微服务各种情况的组件，例如，作为客户端负载均衡的 Ribbon（Eureka 可感知的客户端负载均衡），作为断路器的 Hystrix（也由 NetFlix 开发），等等。这些辅助工具的实现方法的讨论已经超出了本书的范畴，读者可以自行研究 spring-cloud 和其为微服务架构提供的可用功能。

2.2.4 整个架构中的配置外化

配置管理一直是开发人员比较头疼的问题。某些代码中习惯性地耦合了一些配置信息，这种做法非常不好，因为一旦要修改配置就需要重新发布。此外，如果在多个微服务中的某些配置信息相同，如数据库服务器地址等，那么这些信息会在所有服务中产生多个副本。一个比较好的做法是，将配置信息从代码中抽离出来。实现方式是我们在微服务之间共享一个公共区域来存储配置属性值。DevOps 人员可以在所有微服务实例上挂载一块硬盘，使用 NFS 网络文件服务也是一个办法，不同环境有不同的 NFS 网络文件服务器和磁盘。

如果用的是 AWS，可以使用 AWS 的 S3 存储区来做。存储区可以通过设置层级关系来给微服务提供一些定制的属性。每个微服务对应一个不同的文件夹，文件夹内存放



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

该微服务所需的各种不同配置文件。默认约定的存储属性文件的方式是：

```
s3://<bucketName>/<Environment_Name>/<microservice_name>.
```

在这种场景中，完全没有对配置改动的跟踪。为了解决这个问题我们可以引入一些版本管理工具来管理配置。开发人员可以提交代码库中的相关配置改动并在部署微服务的时候由 CI/CD 工具部署，并将这些配置部署到共享路径下。通过观察版本控制的日志，我们可以知道某个版本的微服务具体在使用哪一个配置。因此，关键是要在部署微服务的时候标记配置。使用版本管理工具来维护配置历史之后，还要考虑这个问题：一个组织里面通常会有多个环境，每个环境有其特有的属性。这个问题通常可以通过使用多分支或者多配置代码库来解决，只不过这个时候，CI/CD 工具需要知道每次部署的是哪一个环境，以及相应地应该使用哪个分支或者哪个代码库的配置文件。

因为本书是基于 Spring Boot 的，所以我们可以着重讨论一下 Spring Boot 中的做法。Spring Boot 会按照特殊的顺序来搜索代码中的配置信息，顺序如下。

（1）查找命令行参数。

（2）尝试寻找 `SPRING_APPLICATION_JSON`（某个环境变量或系统属性中内置的 JSON）中的属性。

（3）尝试从默认的 `java:comp/env` 处查找目录服务 JNDI 属性。

（4）尝试查找 `System.getProperties()`。

（5）尝试查找操作系统具体的环境变量。

（6）查找 `RandomValuePropertySources`。

（7）JAR 包内部或者外部的特定内容应用的 properties 配置文件，如 `application-{profile}.properties`。

（8）JAR 包内部或外部的 application.properties 配置文件。

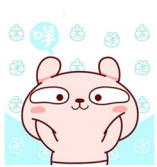
（9）`@PropertySource` 注解。

（10）Spring Boot 会尝试寻找使用 `SpringApplication.setDefaultProperties` 设置的属性值。

因本书篇幅有限，我们不会展开整个查找流程，读者如果有兴趣完全可以自行探索以找到最合适的方式来配置属性。在我们的示例工程中，用的也是`@PropertySources` 注解，我们可以就此举个使用这个注解的例子，通常这个注解用在配置类中，如下：

```
@PropertySources({ @PropertySource("file:/home/user/microserviceA.props")})
```

这个注解会读取从路径为`/home/user/microserviceA.props` 加载的配置文件。一般而言，配置文件中会存放一些键值对类型的属性，然后在应用程序的 Bean 中我们可以通过相应的占位符来使用这些配置属性，像这样：



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@Value("${Connect.ip}")
private String connectionIp;
```

通过这两行代码，Spring 会在前面提到的配置文件中查找到类似 Connect.ip=xx.xx.xx.xx 的属性，然后通过@Value 注解将该属性的值赋给这里的 connectionIp 变量。

在这种类型的配置中，在注解中使用固定路径是有问题的。我们可以使用环境变量来避免这个问题。假设我们生产环境的配置文件路径是/mnt/share/prod/config，在过渡环境中的路径是/mnt/share/stag/conig，则我们的注解可以这样写：

```
@PropertySources({
@PropertySource("file:${CONFIG_PATH}/microserviceA.props") })
```

这里的 CONFIG_PATH 是环境变量的名字，在各个环境可以配置成不同的值。在 Linux 环境下，可以通过 export CONFIG_PATH=/mnt/share/stag/config 来设置。

有时候，在同一个微服务里面，开发人员需要使用多个逻辑上不同的配置文件，这个注解也可以支持从多个文件中加载配置属性。例如：

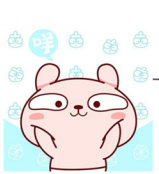
```
@PropertySources({
@PropertySource("file:${CONFIG_PATH}/microserviceA.props"),
@PropertySource("file:${CONF IG_PATH}/connect.props") })
```

在这个例子里，从两个文件中加载配置属性。在任何一个使用配置属性占位符的地方，Spring 会同时查找这两个配置文件的属性。

如果我们使用 Spring Cloud，可以使用 Spring Cloud 提供的配置服务器。配置服务器可以从任何给定的 URL 中读取配置信息。任何一个客户端服务启动的时候都可以知道配置服务所在的位置，客户端服务询问配置服务是否拥有与自身相关的配置属性，然后配置服务器根据请求服务的名称，从配置文件存放的地方查找相关配置属性，并以 JSON 格式返回给客户端。配置服务很容易搭起来，只需要使用@EnableConfigServer 注解就可以。Spring Cloud 中提到的配置服务查找的配置文件 URI 是这样的：

```
server.port=8888
spring.cloud.config.server.git.uri =
  https://github.com/abc/microserviceA-configuration
```

前面提到的配置文件也可以是 YAML 格式的，与我们前面在客户端配置文件中定义配置服务 IP 和端口的方式相同。当客户端服务启动的时候，会访问配置服务来获取配置，配置服务会查找到客户端服务自己配置的名字（spring.application.name），然后在前面提到的 URI 中查找客户端名字所对应的配置文件{clientName}.props。在 Spring Cloud 中，客户端还可以通过@RefreshScope 注解来实现动态加载而无须重新启动客户端



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

服务。使用了这个注解之后，客户端服务会多一个端点<service endpoint>/refresh，通过访问该端点[①]就可以实现动态加载配置，重建 Spring Bean 了。

在本节中，我们讨论了各种实现配置外化的方式，但是只提到了几种，还有很多留给读者自己去探索。使用配置服务，有助于在数百个服务同时运行时动态地改变环境并共享公共属性。但是，却不可避免地增加了开发人员的工作量，因为多了一个配置服务器需要维护，而且要保证这个关键的配置服务随时可用。你还可以在配置服务器前面加一些缓存，来规避配置服务出错。API 网关模式也能解决一些微服务配置外化的问题，下一节我们会详细讨论如何使用 API 网关模式。

2.3　API 网关及其诉求

现如今的动态网站上，通常会在一个页面展示很多信息。例如，一个成功下单的页面可能需要同时展示购物车详情和客户地址信息，为了展示这些信息，前端网页需要发多个不同的请求来查询客户详情服务和订单详情服务。图 2-4 是一个简单的同一页面展示多个服务的例子。因为微服务中每个服务只关心自己的领域，所以为了展示众多的信息，不可避免地要在同一个页面中发很多 API 请求来组合信息，而这往往会显得太过啰唆。

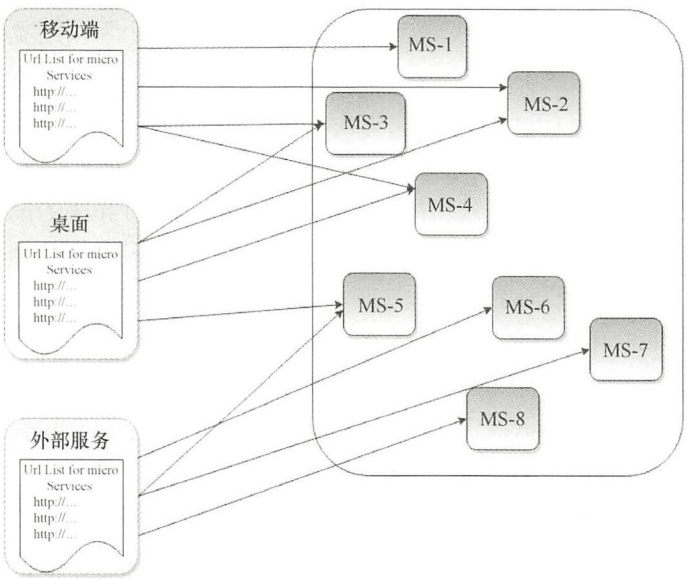


图 2-4

① 该端点需要使用 POST 方法。——译者注

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

另外一个问题是，某些微服务会使用特殊的传输协议，以至于某些外部消费者并不能直接使用这种协议与某个微服务通信。移动端由于屏幕较小，所需的数据可能与网页 API 请求所需的数据不同。例如，开发人员可能想返回给移动端少量数据，或者让移动端使用不同版本的 API 等。因此，你可能会遇到这么个问题：每个客户端在访问不同的网络服务并且与网络服务保持联系，开发人员必须保证服务向后兼容，因为 API URL 被嵌入了移动端 App 等客户端中，不便于修改。

这就是 API 网关想要解决的诉求。前面提到的所有这些问题都可以通过 API 网关来解决。API 网关在 API 消费者和 API 服务器之间扮演了中转代理的角色。为了解决第一个组合请求的问题，我们可以只定义一个总的 API，如叫`/successOrderSummary`，这个请求会直接发给 API 网关。API 网关作为消费者发起订单和用户信息的请求之后，分别得到订单和用户详情，然后将两个结果结合之后一并返回给客户端。因此，在这种情况下 API 网关扮演了一个汇总 API 的角色，它会在内部调用多个 API 来保证对外只暴露一个 API。图 2-5 展示了 API 网关的工作模式。



图 2-5

除此之外，API 网关还解决了其他很多问题，下面我们来一一探讨。

2.3.1 认证鉴权

API 网关可以用来做外部 API 请求的权限验证。只要请求通过了 API 网关的鉴权，内部调用就无须进行任何权限检查了。VPC 内部的请求也可以去除安全检查来降低网络



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

延时。如此一来，开发人员只需专注于业务逻辑开发，而不用顾及安全问题了。

2.3.2　不同协议

某些时候，各个微服务内部通信可以使用不同的协议来对话，如 Thrift 调用、TCP、UDP、RMI、SOAP 等。对于整个微服务系统的客户端来说，一般都是基于 Rest 的 HTTP 调用。客户端是用 HTTP 协议访问到 API 网关的，然后 API 网关按照各个服务需要的协议来发起内部调用，对最终结果进行组装之后按照客户端要求的协议返回数据，大多数时候客户端都采用 HTTP 协议。

2.3.3　负载均衡

API 网关还可以用作负载均衡器来高效处理并发请求。API 网关可以跟踪它发往某个具体服务不同节点的请求的负载量，根据具体的负载量在特定服务的节点之间智能地做负载均衡。有了 NGINX Plus 之后，NGINX 可以作为一个不错的 API 网关的替代品，它有很多可以处理 API 网关面临的常用问题的功能。

2.3.4　请求转发（包括服务发现）

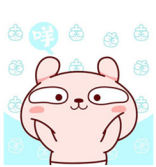
网关的一个重要作用是减少客户端和微服务之间的通信次数。所以某些时候如果客户端需要的话，会并行访问一些微服务。从客户端来看，仍然只是一次调用。

网关可以同时请求所有需要的服务，然后等待所有这些服务的返回值。在得到所有服务的返回值后，网关会将结果组合到一起并发回给客户端。使用响应式的微服务设计可以帮你实现这一点。

结合服务发现之后，还可以有更多的功能。例如，API 网关可以判断服务的主节点和从节点。同样，在数据库方面，可以做读写分离，保证所有的写请求都发往主库，读操作都来自从库。读写分离只是一个基本的用法，用户还可以根据 API 网关提供的原信息做很多别的事情。网关可以记录服务实例每个节点的基础响应时间，对于某些高优先级的 API 调用，可以路由到响应更快的节点来处理。最后再总结一下，有了 API 网关这个基础，你可以用不同的办法做很多事情。

2.3.5　响应转换

作为所有 API 的第一个也是唯一一个入口，API 网关可以获知客户端的类型，例如，是来自移动端、网站，还是来自其他一些外部消费者等。API 网关可以根据客户端的不同，访问不同的内部服务，然后按照客户端的需要和配置返回不同的数据。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.3.6 断路器

为了解决局部失败的问题，API 网关使用了一种称为**断路器**的技术。某一个服务的调用失败往往会导致整个服务调用链的失败，引起连锁反应。API 网关可以给每一个微服务设置一个阈值，如果任何一个服务的调用失败数量超出了阈值，断路器会将该服务设置为断路状态，在一个可配置的时间内，不再发起对该服务的调用。Netflix 公司的 Hystrix 就是一种高效的断路器，它的默认阈值是 5 s 内 20 个失败请求。开发人员还可以针对断路情况做一些回退处理，如虚设一些返回值等。一旦 API 又重新恢复正常了，网关会终止断路状态，继续访问该服务。

2.3.7 API 网关的优劣性

使用 API 网关有其自身的优势和劣势。在前面几节中，我们已经阐述了所有使用 API 网关的好处。下面简要地对 API 网关的优点进行一个总结：

优点如下：

- 微服务可以专注于业务逻辑本身；
- 客户端可以通过一次请求得到所有需要的数据；
- 可以通过 API 网关来做认证、日志和监控；
- 可以解决客户端和微服务使用完全不同协议的问题；
- 可以根据客户端需要对结果进行一些裁剪；
- 可以处理局部失败问题。

除了上面的这些优点之外，API 网关也有一些需要权衡的劣势。

劣势如下：

- 如果 API 网关做的事情过多，可能会影响性能；
- 要想使用 API 网关必须先有服务发现；
- 有些时候，API 网关会成为单点失败的风险点，因为它承载了所有 API 的入口；
- 需要额外增加管理路由的任务量；
- 增加了额外的网络请求，也就是说先从 API 网关走了一道；
- 总的来说，API 网关增加了系统复杂性；
- 在 API 网关中实现太多逻辑的话，会导致一些依赖问题。

所以，在选用 API 网关之前，应该先考虑好这些方面。在系统中引入 API 网关毫无疑问会增加开销。在开始实施之前，建议先认真评估一下使用 API 网关需要的成本、开销、管理成本以及从中获得的收益。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.4　API 网关的例子

在这个例子里，我们将展示示例产品的页面，产品信息来自某个服务。这个例子可以在多个方面进行扩展。我们的重点是借这个例子展示 API 网关的工作模式，所以这个例子会做得比较简单。

我们选用 Netflix 的 **Zuul** 作为 API 网关。Spring 内部也实现了 Zuul，于是我们可以从 Spring Boot 工程开始创建这个例子。对于这个 API 网关的实现，我们会使用 `http://start.spring.io/`来生成一个初始化模板。这个 Spring 提供的 Spring 初始化器是帮助初学者生成基本 Spring Boot 代码的工程。用户只需要设置少量配置参数，然后点击 **Generate Project** 按钮就可以下载初始工程代码了。如果用户想设置一些更详细的项目参数，可以点击 **Switch to the full version** 链接来展开更多配置项，如图 2-6 所示。



图 2-6

对 Zuul 而言，用户需要点击 **Advance**，然后选中 **zuul** 勾选框，再点击 Generate Project 按钮，就可以开始下载示例工程了。

如果你打开下载下来的项目中的 `pom.xml`，你可以看到其中所包含的 `zuul` 依赖，如下所示：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

现在在主应用类相同的包下创建一个控制器类，代码如下：

```
@SpringBootApplication
@RestController
public class ProductDetailConrtoller
 {
    @Resource
    ProductDetailService pdService;

    @RequestMapping(value = "/product/{id}")
    public ProductDetail getAllProduct( @PathParam("id") String id)
    {
        return pdService.getProductDetailById(id);
    }
}
```

上面的代码中假设了 pdService 这个 bean 可以与 Spring 数据代码库（Spring data repository）交互来根据请求中的产品 ID 得到其详细信息。另外一个假设是这个服务运行在 10000 端口。如果一切正常的话，访问 http://localhost:10000/product/1 可以看到返回的一些 JSON 数据。

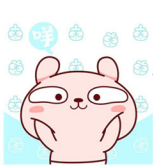
对于 API 网关，我们将另建一个支持 Zuul 的 Spring Boot 应用。激活 Zuul 只需要添加一个简单的@EnableZuulProxy 注解。

下面是一个使用了 Zuul 代理的代码：

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayExampleInSpring
{
    public static void main(String[] args)
    {
       SpringApplication.run(ApiGatewayExampleInSpring.class, args);
     }
}
```

剩下的所有事情都可以在配置中搞定。在 API 网关的 application.properties 文件中，添加如下内容：

```
zuul.routes.product.path=/product/**
zuul.routes.produc.url=http://localhost:10000
ribbon.eureka.enabled=false
server.port=8080
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

有了这些配置之后，我们定义了这样一些规则：对于接收到的任何一个 URL 格式为 /product/xxx 的请求，将该请求转发给 http://localhost:10000。对于外部的世界而言，可能的 URL 是 http://localhost:8080/product/1，这个请求会在内部被转发到 10000 端口。如果我们在产品详情微服务的配置中，将 spring.application.name 设置为了 product，那么在这里就不需要在 API 网关中配置 URL 路径属性了（zuul.routes.product.path= /product/**），因为 Zuul 默认会把 application.name 也就是/product 添加到 URL 请求中。

这个例子中的 API 网关虽然不怎么智能，但是功能已经很强大了。有了 Zuul 中定义的属性，如路由、过滤器和缓存等，就能建立一个功能更加强大的 API 网关了。

贯穿本书的示例应用

前面已经提过，我们会在本书中贯穿一个**信用风险评估引擎**的应用。风险引擎其实是一堆计算风险的规则，在对输入参数进行一些计算之后，产出是一个风险等级。我们加上了信用这个前缀，你可以想象为它是在某些信用事务中做信用的风险评级的。信用风险可以有多种类型，如借贷额度、共同基金、债券、股票等。借钱给机构或者个人的时候，始终要考虑风险这个因素。

信用风险评估引擎需要提取一些个人或者组织的以往和当前的信用信息，然后根据定义的计算规则，给出信用评分。根据信用评分，评估引擎决定是否给这个用户提供信用贷款。银行或者金融机构主要靠借钱给借款人来赢利，在做某些大型投资的时候，也会使用一些类似的风险评估引擎来评估风险。实际系统中需要尽可能多的参数来让分析评估尽可能精准。

言归正传，在这个的信用风险引擎项目中，我们会专注于个人贷款业务。假设个人用户需要一笔贷款来购买某个产品，产品价格从 5 000 到 40 000 不等。在 **ZestMoney** 网站上有很多复杂的规则和算法来计算借款时的风险等级。但是在本书的这个示例项目中，只专注于主要的 3～4 个参数。基于这些参数，信用风险评估引擎会判断出是应该接受还是拒绝某一笔贷款。

在本书的课程中，我们将会开发下面这些服务。

- 用户注册。
- 财务详情：
  - 收入详情；
  - 债务详情。
- 信用评分。

这些服务的更多细节会在后面的相关章节中介绍。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.5 用户注册微服务的开发

本节主要讨论如何启动配置服务和主应用。

配置服务器

与前面的例子中一样，还是采用 start.spring.io 来创建一个 Spring Boot 应用。这一应用的群组名是 com.practicalMicroservices。首先创建一个配置服务器，来给项目中不同的服务提供配置信息。创建一个名为 configServer 并且选中 cloud starter 选项的 Spring Boot 应用。从 start.spring.io 下载下来的工程，其目录结构如图 2-7 所示。

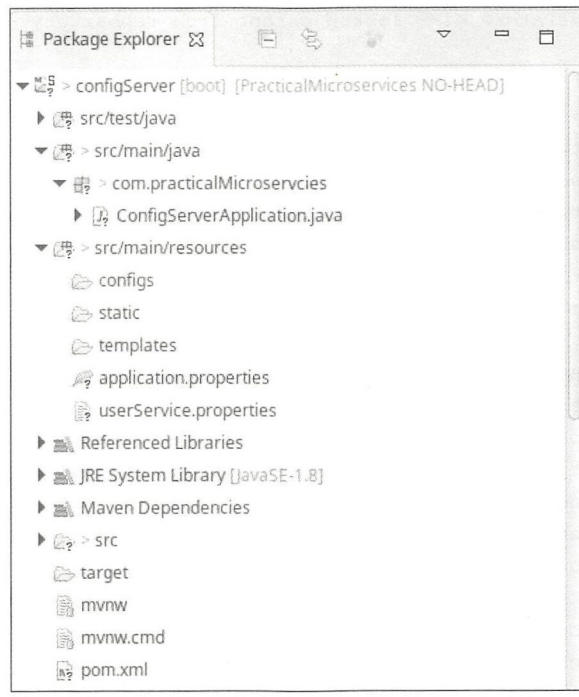
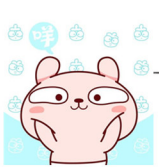


图 2-7

pom.xml 文件内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.practicalMicroservcies</groupId>
    <artifactId>configServer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>configServer</name>
    <description>Configuration Server to provide configuration for
    different services</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-
        8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
        8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-server</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>
.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

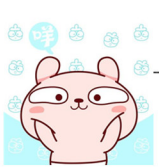
我们在 POM 文件中添加了一个 cloud-started 依赖。此外，我们还会有一个叫 Camden 的客户端。在本书撰写时，Spring Boot 最稳定的版本是 1.4.1，所以这段代码里面使用这个版本。

主应用类代码如下：

```
package com.practicalMicroservcies;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    }
}
```

这里再没有什么别的内容了，唯一的改动是@EnableConfigServer 注解。这个注解告诉 Spring 将当前应用视为配置服务器。现在下一步是定义配置服务器应该在哪一个端口上运行，同时还要定义不同微服务查询的不同属性文件。

为了给配置服务器定义端口，需要在 src/main/resources/application.properties 文件中添加值，如下所示：

```
server.port=8888
spring.profiles.active=native
spring.cloud.config.server.native.searchLocations=classpath:/
```

这些属性告诉 Spring 服务器应该运行在哪个端口上，这里指定的端口是 8888。下面的两个属性定义了配置文件的地址，这里指定的是 native 而不是 Git 的代码库位置，因此配置服务器会查找 native 性能分析文件中制定的搜索位置。下一个属性 spring.cloud.config.server.native.searchLocations 告诉 Spring 所有的配置文件应该在类路径下。在这里，所有的配置文件都放在 resources 文件夹下，这也是默认的类路径。

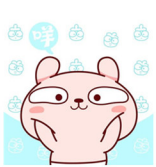
下一步在 resources 目录下创建一些新的属性文件。现在先创建一个名为 userService.properties 的文件，其内容如下：

```
Connect.database=practicalMicroservice
spring.datasource.url=jdbc:mysql://localhost:3306/practicalMicroservice
spring.datasource.username=password
spring.datasource.password=password
# To generate SQL optimized for a Mysql
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
# Naming strategy, it can resolve the issue of uppercase table or column name
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
```

这个配置文件定义了 userService 所需的一些配置属性。至此配置服务器代码已经完成了。在终端上运行 mvn spring-boot:run 就可以启动应用程序了。通过一个 curl 命令就能看到属性文件的结果，也就是下面的命令：

```
curl http://localhost:8888/userService/default
```

- localhost:8888：这是服务器地址和端口。
- userService：这是我们需要获取属性的服务名称。
- default：这是默认配置文件的名字。如果没有专门定义，会使用 default 作为名字。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这个命令的结果如下：

```
{
  "name": "userService",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
     "name": "classpath:/userService.properties",
     "source": {
       "spring.jpa.show-sql": "true",
       "spring.datasource.username": "root",
       "spring.jpa.hibernate.naming-strategy":
       "org.hibernate.cfg.ImprovedNamingStrategy",
       "spring.datasource.password": "root123",
       "Connect.database": "practicalMicroservice",
       "spring.jpa.properties.hibernate.dialect":
       "org.hibernate.dialect.MySQL5Dialect",
       "spring.datasource.url":
       "jdbc:mysql://localhost:3306/practicalMicroservice"
     }
    }
  ]
}
```

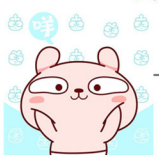
在<source>标签下，就能看到我们定义在 userService.properties 文件中的所有属性了。

```
User service
```

用户服务是什么呢？它负责所有与用户相关的操作，包括用户注册、修改、查找和删除等。相应的会有 controller、repository、service 和 model 等包，还会有一个控制器来处理获取用户详情和用户地址的请求。服务包中主要包含添加、修改、删除用户详情和添加用户数据等功能。这里的删除并不会真的从数据库中删除记录，而是通过数据库中一个 deleteOn 字段来标记已删除。

表结构

用户服务会用到两张表。一张表用来记录用户的个人数据，如名字、唯一 ID 号、出生日期、性别等。第二张表用来保存用户的地址信息。我们会使用 flywaydb 和下面



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的 SQL 文件来创建模式（schema）和表结构。**Flyway** 是一个开源的数据库迁移工具，Flyway 有相应的 Maven 插件可以完美契合 Spring Boot。在 Spring Boot 中使用 Flyway 需要在 POM 文件中添加下面的依赖：

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>4.0.3</version>
</dependency>
```

Flyway 数据库的约定是，会查找 `src/main/resources/db/migration` 路径下的名字格式为 `V1_0__DESCRIPTION.sql` 的文件。Flyway 数据库会在数据库中的存储模式的修订。Flyway 通过读取模式版本可以知道哪些迁移脚本已经成功执行过了，哪些是新加的要运行。下面我们通过 Flyway 创建 `user_details` 和 `addresses` 表。创建 `user_details` 表的 SQL 代码如下：

```
CREATE TABLE `user_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `first_name` varchar(250) NOT NULL,
  `middle_name` varchar(250) DEFAULT NULL,
  `last_name` varchar(250) DEFAULT NULL,
  `created_on` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  `deleted_on` TIMESTAMP DEFAULT NULL,
  `leagal_id` varchar(10) NOT NULL,
  `date_of_birth` TIMESTAMP NOT NULL,
  `gender` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  KEY `user_details_userId_DeletedOn` (`user_id`,`deleted_on`),
  KEY `user_id_deleted` (`deleted_on`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

创建 `addresses` 表的 SQL 代码如下，其中 `table.user_id` 是关联的 `user_details` 表中的 ID：

```
CREATE TABLE `addresses` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `city` varchar(25) NOT NULL,
  `address_line_1` varchar(250) NOT NULL,
  `address_line_2` varchar(250) DEFAULT NULL,
  `pincode` char(6) NOT NULL,
  `created_on` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
  `deleted_on` TIMESTAMP DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  KEY `addresses_user_id_DeletedOn` (`user_id`,`deleted_on`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;
```

与在配置服务中的做法类似，先从 `start.spring.io` 下载一个名为 `userService` 的 Spring Boot 应用。这次会一并选上 Spring 数据和 MySQL 连接器来创建一个新项目。这里使用的是 Spring 工具套件（STS）的 3.8.3 版本。

在下载完代码并解压到某个特殊文件夹之后，可以在当前目录下运行 `mvn eclipse:eclipse` 命令来产生与 Eclipse 相关的文件。此后，就可以轻易将项目导入 Eclipse 或者基于 Maven 的 STS 中去了。在将项目导入自己喜欢的 IDE 中去之后，会看到下面这样的文件目录。最终项目的文件结构应该与图 2-8 所示的结构类似。



图 2-8

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

为了达到这个效果，需要从 POM 文件着手进行修改。POM 文件应该包含如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

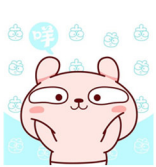
    <groupId>com.practicalMicroservcies</groupId>
    <artifactId>userService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>userService</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8
        </project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
        </project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
```



仅供非商业用途或交流学习使用

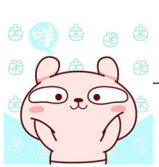
仅供非商业用途或交流学习使用


```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-core</artifactId>
            <version>4.0.3</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

这个 POM 文件中有些新的配置，如 Flyway 依赖、spring-data-jpa 和 MySQL 连接器等。下面需要往主应用类中添加下列代码：

```
package com.practicalMicroservcies;

import org.flywaydb.core.Flyway;
import org.springframework.beans.factory.annotation.Value;
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class UserServiceApplication {
    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String userName;
    @Value("${spring.datasource.password}")
    private String password;
    @Value("${Connect.database}")
    private String database;

    @Bean(initMethod = "migrate")
    public Flyway flyway() {
        /**
        As flyway Db need url seperetaly that is why we extracting only
        url from  given string
        */
        String urlWithoutDatabaseName=url.substring(0,url.lastIndexOf("/"));
        Flyway flyway = new Flyway();
        flyway.setDataSource(urlWithoutDatabaseName, userName, password);
        flyway.setSchemas(database);
        flyway.setBaselineOnMigrate(true);
        return flyway;
    }

    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

这里使用了一些在配置服务器中定义的属性，还需要定义清楚这个服务是如何从配置服务器获取相关属性的。为了做到这一点，请在 resources 文件夹中添加一个 bootstrap.properties 文件，并写入下列内容：

```
spring.application.name=userService
spring.cloud.config.uri=http://localhost:8888
```

这两个属性简单易懂。spring.application.name 键定义的是当前服务的名称，这也是与配置服务器通信的时候所使用的服务名，配置服务器会按照该服务名查找属于该服务的属性文件并予以返回。Bootstrap 属性会在整个项目初始化之前被执行，因此这个定义应用名的属性会被最先执行。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

下面这个以 spring.datasource 开头的属性是十分重要的，Spring Boot 会使用 datasource 开头的 3 个属性自动创建一个数据源而无须写任何代码。connect.database 不是 Spring Boot 的内部属性，可以使用任何名字来定义数据库的名字。

在 flywayDb bean 中，我们设置了这个版本 Flyway 数据库下的模式（schema）名。如果模式不存在，Flyway 会创建这个模式。

因此，如果在终端运行 mvn spring-boot:run，应该能看到在 MySQL 中创建的数据库和表。

下面在 entity 包下给这两张表创建模型类，一个是 Address，另一个是 UserDetail。在 Address.java 文件中插入如下内容：

```
package com.practicalMicroservcies.entity;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "addresses")
public class Address implements Serializable {

    /**
     * Serial Id for serialization
     */
    private static final long serialVersionUID = 1245490181445280741L;
    /**
     * unique Id for each record.
     */

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;

    /**
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
     * User Id , unique for every user
     */

    @Column(name = "user_id", nullable = false, unique = true, length = 36)
    private String userId;

    /**
     * City, Staying city of user
     */
    @Column(name = "city", nullable = false, length = 25)
    private String city;

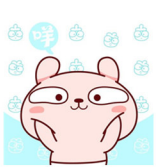
    /**
     * Address Line 1 for User Address.
     */
    @Column(name = "address_line_1", nullable = false,  length = 250)
    private String addressLine1;

    /**
     * Address Line 2 for User Address.
     */
    @Column(name = "address_line_2", nullable = false,  length = 250)
    private String addressLine2;

    /**
     * Pincode for user City
     */
    @Column(name = "pincode", nullable = false,  length = 36)
    private String pinCode;

    /**
     * Date on which user is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP", length = 0)
    private Date createdOn;

    /**
     * Date on which user is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT
    NULL", length = 0)
    private Date deletedOn;
}
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这里还应该给类变量创建一些 getter 和 setter 方法[①]。用户也可以选择重载模型类的 toString 方法，代码如下：

```
@Override
public String toString() {
    return "Address [id=" + id + ", userId=" + userId + ", city=" +
    city + ", addressLine1=" + addressLine1 + ", addressLine2=" +
    addressLine2 + ", PinCode=" + pinCode + ", createdOn=" +
    createdOn + ", deletedOn=" + deletedOn + "]";
}
```

除了模型类，还需要创建一个代码库（repository）类，如下所示：

```
package com.practicalMicroservcies.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.Address;

public interface AddressRepository extends JpaRepository<Address, Integer>
{
    Address findByUserId(String userId);
}
```

UserDetail 也需要一个实体类，如下所示：

```
package com.practicalMicroservcies.entity;

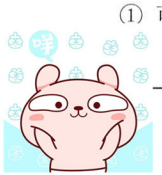
import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "user_details")
public class UserDetail implements Serializable {
```

① 可以使用 IDE 快捷键或 Lombok 等辅助工具。——译者注



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
/**
 *
 */
private static final long serialVersionUID = 4804278804874617196L;

/**
 * unique Id for each record.
 */

@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "id")
private int id;

/**
 * User Id , unique for every user
 */

@Column(name = "user_id", nullable = false, unique = true, length = 36)
private String userId;

/**
 * First Name of User
 */
@Column(name = "first_name", nullable = false, unique = true, length = 250)
private String firstName;

/**
 * Last Name of User
 */
@Column(name = "last_name", nullable = false, unique = true, length = 250)
private String lastName;

/**
 * Middle Name of user
 */
@Column(name = "middle_name", nullable = false, unique = true, length = 250)
private String middleName;

/**
 * Legal id Of user( could be social security number etc)
 */
@Column(name = "legal_id", nullable = false, unique = true, length = 20)
private String legalId;
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    /**
     * Gender of user
     */
    @Column(name = "gender", nullable = false, unique = true, length = 20)
    private String gender;

    /**
     * Date of birth of user
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "date_of_birth", columnDefinition = "TIMESTAMP", length = 0)
    private Date dateOfBirth;

    /**
     * Date on which user is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP", length = 0)
    private Date createdOn;

    /**
     * Date on which user is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT NULL",
    length = 0)
    private Date deletedOn;
}
```

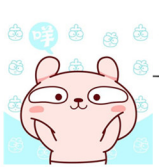
UserDetail 类的所有实例变量也同样需要一些 getter 和 setter 方法。UserDetail 的 toString 方法可以重载成下面这样：

```
@Override
public String toString() {
    return "UserDetail [id=" + id + ", userId=" + userId + ",
    firstName=" + firstName + ", lastName=" + lastName + ",
    middleName=" + middleName + ", legalId=" + legalId + ",
    gender=" + gender + ", createdOn=" + createdOn + ", deletedOn="
    + deletedOn + "]";
}
```

该实体的代码库类也是必需的：

```
package com.practicalMicroservcies.repo;

import org.springframework.data.jpa.repository.JpaRepository;
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
import com.practicalMicroservcies.entity.UserDetail;

public interface UserDetailRepository extends JpaRepository<UserDetail,
Integer> {
    UserDetail findByUserId(String userId);
}
```

同时需要一个 service 类来处理这些代码库：

```
package com.practicalMicroservcies.service;

import java.util.Date;
import java.util.UUID;

import javax.annotation.Resource;
import javax.transaction.Transactional;

import org.springframework.stereotype.Service;

import com.practicalMicroservcies.entity.Address;
import com.practicalMicroservcies.entity.UserDetail;
import com.practicalMicroservcies.repo.AddressRepository;
import com.practicalMicroservcies.repo.UserDetailRepository;

@Service
@Transactional
public class UserDetailServices {

    @Resource
    AddressRepository addressRepo;

    @Resource
    UserDetailRepository userRepo;

    public void saveAddress(Address address) {
        addressRepo.save(address);
        System.out.println("User Saved!");

    }

    public void saveUser(UserDetail userDetail) {
        userRepo.save(userDetail);
        System.out.println("User Saved!");

    }
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    public Address getAddress(UUID userId) {
        Address returnAddressObject =
        addressRepo.findByUserId(userId.toString());
        return returnAddressObject;

    }

    public UserDetail getUser(UUID userId) {
        UserDetail userObjectToRetrun =
        userRepo.findByUserId(userId.toString());
        System.out.println("User Saved!");
        return userObjectToRetrun;

    }

    public void deleteUser(UUID userId) {
        Address addressObject =
        addressRepo.findByUserId(userId.toString());
        addressObject.setDeletedOn(new Date());
        addressRepo.saveAndFlush(addressObject);
        UserDetail userObject =
        userRepo.findByUserId(userId.toString());
        userObject.setDeletedOn(new Date());
        userRepo.saveAndFlush(userObject);
        System.out.println("User Deleted!");

    }

}
```

最后，我们需要一个控制器类来处理来自前端的请求。这个服务中的用户 ID 是一个 UUID，该 ID 由前端产生。如此一来，前端或者消费者就可以使用相同的用户 ID 并行访问该微服务了，代码如下：

```
@RestController
@RequestMapping("/PM/user/")
public class UserController {

    @Resource
    UserDetailServices userService;

    @Resource
    ObjectMapper mapper;
}
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这里，我们还需要两个服务，一个是 UserDetailServices，用来做与用户相关的操作，以及 JSON 和 Java 对象之间的相互转换。

此外，还需要一个方法来负责在数据库中给某个用户添加新地址：

```
@RequestMapping(method = RequestMethod.POST, value = "{userId}/address",
produces = "application/json", consumes = "application/json")
public ResponseEntity<String> createUserAddress(@RequestBody
Address address, @PathVariable("userId") UUID userId) {
    logger.debug(createUserAddress + " Address for user Id " +
    userId + " is updated as " + address);
    address.setUserId(userId.toString());
    userService.saveAddress(address);
    return new ResponseEntity<>(HttpStatus.CREATED);
}

// Below  Method is responsible for creating a user.

public static final String createUser = "createUser(): ";

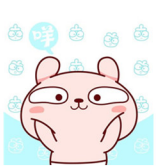
@RequestMapping(method = RequestMethod.POST, value = "{userId}",
produces = "application/json", consumes = "application/json")
public ResponseEntity<String> createUser(@RequestBody UserDetail
userDetail, @PathVariable("userId") UUID userId) {
    logger.debug(createUser + " creating user with Id " + userId +
    " and details : " + userDetail);
    userDetail.setUserId(userId.toString());
    userService.saveUser(userDetail);
    return new ResponseEntity<>(HttpStatus.CREATED);
}
```

下面的代码会处理按照用户 ID 删除用户的请求：

```
@RequestMapping(method = RequestMethod.DELETE, value = "{userId}",
produces = "application/json", consumes = "application/json")
public ResponseEntity<String> deleteUser(@RequestBody UserDetail
userDetail, @PathVariable("userId") UUID userId) {
    logger.debug(deleteUser + " deleting user with Id " + userId);
    userService.deleteUser(userId);
    return new ResponseEntity<>(HttpStatus.CREATED);
}
```

下面的方法负责按照 ID 获取用户详情，这是一个 GET 请求：

```
@RequestMapping(method = RequestMethod.GET, value = "{userId}",
produces = "application/json", consumes = "application/json")
public ResponseEntity<UserDetail> getUser(@PathVariable("userId")
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
UUID userId) {
    logger.debug(getUser + " getting information for userId " + userId);
    UserDetail objectToReturn = userService.getUser(userId);
    if (objectToReturn == null)
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    else
        return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
}
```

下面这个方法负责从数据库中获取指定用户 ID 对应的地址列表：

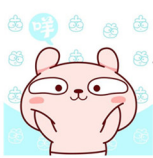
```
@RequestMapping(method = RequestMethod.GET, value =
"{userId}/address", produces = "application/json", consumes =
"application/json")
public ResponseEntity<Address> getAddress(@PathVariable("userId")
UUID userId) {
    logger.debug(getAddress + " getting address for user Id: " +
    userId);
    Address objectToReturn = userService.getAddress(userId);
    if (objectToReturn == null)
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    else
        return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
}
```

该控制器暴露的 URL 有下面这些：

```
Post http://localhost:8080/PM/user/<user_Id>
Post http://localhost:8080/PM/user/<user_Id>/address
Get Post http://localhost:8080/PM/user/<user_Id>
Get Post http://localhost:8080/PM/user/<user_Id>/address
Delete Post http://localhost:8080/PM/user/<user_Id>
```

至此，我们可以测试一下这个简单的用户服务了。先在不同的终端通过 `mvn spring-boot:run` 命令启动配置服务器和用户详情服务，我们会看到，Flyway 自动创建数据库（数据库脚本如图 2-9 所示）后可以使用一个 rest 客户端（如 postman 或者 curl 等）来测试 URL 是否能正常工作。对于地址实体，可以用下面这样一个 JSON 值尝试一下发一个 `POST` 请求：

```
{
  "userId": "93a52ce4-9331-43b5-8b56-09bd62cb0444",
  "city": "New York",
  "addressLine1": "4719 Fermun Road",
  "addressLine2": "Yorktown Heights",
  "pinCode": "10004"
}
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

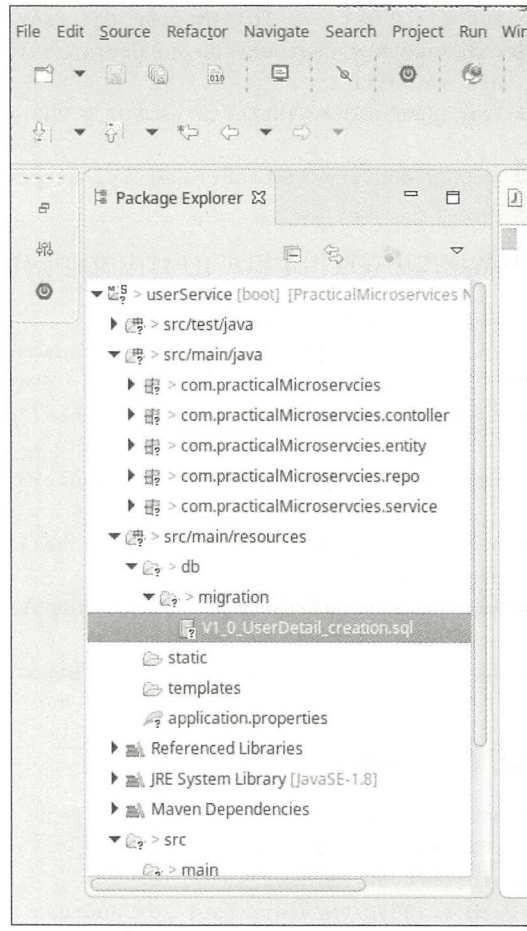


图 2-9

要创建用户，可以使用下面这样一个 JSON：

```
{
  "userId": "93a52ce4-9331-43b5-8b56-09bd62cb0444",
  "firstName": "John",
  "lastName": "Montgomery",
  "middleName": "Allen",
  "legalId": "B053-62-64087",
  "gender": "1",
  "dateOfBirth": "2010-10-27T11:58:22.973Z",
}
```

以上是一个简单的服务作为我们的示例。关于安全和测试部分的内容会在后续章节



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

继续讨论。

2.6 小结

本章介绍了服务发现、外化配置和 API 网关等概念，我们还给每个概念举了一些简短的例子。因为本书会以 Spring Boot 作为编码技术基础，所以这些例子也是基于 Spring Boot 实现的。与此同时，我们还定义了第一个微服务——用户注册服务。这只是一个起步，后续会围绕这个服务进一步展开安全、通信等方面的讨论。第 3 章将讨论微服务之间是如何进行通信的，同时还会创建下一个微服务，即财务服务。我们还会引入用户注册微服务的通信。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 3 章
微服务端点之间的通信

第 2 章主要讨论了服务发现和 API 网关模式。本章将讲解如何发现微服务，但重点是挖掘在协议层面微服务实际上是如何通信的。微服务之间的内部通信模式有许多选择。一旦选定了一个符合自身需求的通信模式，就要开始着手学习将会用到的协议了。常见的通信协议有 SOAP、REST、XML-RPC 等。本章还会就微服务间通信的主流模式进行讲解，最后会以一种模式为例讲解微服务间的通信。

本章将会覆盖下列主题：

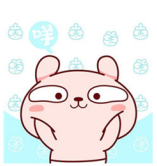
- 微服务间应该如何通信；
- 编制（orchestration）和编排（choreography）；
- 同步通信和异步通信；
- 使用消息代理的事件驱动微服务；
- 财务微服务的开发。

3.1 微服务间应该如何通信

在当今的软件系统中，一个安卓应用的页面或者一个网页就可以展示大量不同种类的数据。如果后台是基于微服务的，那么这些不同种类的数据不可能是由单一微服务生成的，一般需要多个微服务协同才能产生这些不同种类的数据。要实现多个微服务协同工作就免不了需要微服务间的频率通信。

在处理单体应用时，整个代码库都是同一种语言并放在一个地方的。单体应用中各个模块之间通过函数和方法调用来进行通信。但是，在微服务架构中，这是行不通的，因为每一个微服务都是一个单独运行的进程，各个微服务的上下文也完全不同，甚至可能连运行的硬件环境都有所不同。我们可以从不同的视角解读微服务之间的内部通信模式。

如果从控制流和通信流的视角来看，微服务之间的通信可以分为编制和编排两种。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.2　编制和编排

假设有一个团队，大家有一个共同的目标。这个时候可能的情形有两种：第一种情形是团队中有一个领导，他是团队的核心大脑，他对整个团队的能力非常了解，团队中每个个体的工作都由这个领导来分工协调。如果两个工作有依赖关系，就由这个领导来负责在被依赖的工作完成时再开始安排后续工作；另外一种情形是团队中每个人都是自组织的，相互之间有需要的时候自行沟通（通过电子邮件、电话、即时聊天或者其他媒介），并不需要一个专门的领导来安排。在第一种情形下，你需要专门雇一个领导来管理团队，某些时候找个适合团队的领导也挺难的，而且还必须为这个领导支付一笔可观的工资。第二种情形也有一些劣势，例如，你在做完某件事之后，需要广而告之让所有人都知道。在第一种情形中，要知道团队的进度可以随时问领导，但是在第二种情形中，你需要与每一个团队成员交流才能知道整个团队的确切状态。

这两种情形对应的模式分别称为编制和编排，下面我们来深入讨论这两种模式。

3.2.1　编制

orchestration（编制）这个词又指管弦乐编曲，编制这种模式是从管弦乐队的指挥这种角色衍生而来的。指挥的工作是通过一些标志来安排各个微服务的工作，每个微服务会相应地做出回应。我们可以类比成一个音乐家团队，团队中的每个人都知道自己要做的事，也能做得很好。但是这些音乐家是如何知道应该按照什么节奏和曲调演奏，以及在什么时候开始自己的表演呢？这都是通过指挥挥舞指挥棒来控制的，通过某些约定的手势，每位音乐家就知道什么时候演奏哪个乐器，最终完成一次完整的演出，如图 3-1 所示。



图 3-1

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在微服务模式中也使用了同样的概念。微服务中需要一个调度服务来扮演管弦乐队（这里类比的是多个微服务组成的团体）中指挥的角色。这个调度服务会触发并请求各个微服务开始相应的动作。这个时候所有的微服务都服务于这个单一的业务目的。

这个中间微服务通过给相应微服务发送消息的方式来进行调度。这些消息既可以通过 HTTP 请求发起也可以通过套接字请求发起。需要指明的一点是，这里的请求既可以是同步的，也可以是异步的。一般来说，这种模式中个体微服务是不会相互通信的。这种模式下修改工作流会比较简单。在编制模式中，协议类型是不受限的，调度服务能够支持多种不同服务需要的多种不同协议。例如，服务 A 的通信协议是 HTTP，服务 B 的协议是 XML/RPC 等。在第 2 章中探讨的智能 API 网关模式就可以起到这里调度服务的作用。

这种方式也是很有挑战的，这种模式下调度服务的权力过于集中，所有服务的启动都始自这个中心调度服务。这种较为集中的调服服务的存在并不是一个很好的微服务设计。调度服务中的逻辑增加了很多服务之间的依赖，例如，假设调度服务有一个职责是在服务 A 响应之后开始调用服务 B，那么这个时候服务 B 其实是依赖于服务 A 的。

在下面的例子中，调度服务首先会检查消费历史和银行明细，然后请求决策服务做出决策，如图 3-2 所示。这个时候很容易可以知道其贷款的申请进度，但是由于核心逻辑现在都放在贷款服务（也就是调度服务）中，随着系统的逐渐扩展，这个起调度作用的贷款服务会对整个系统产生影响。

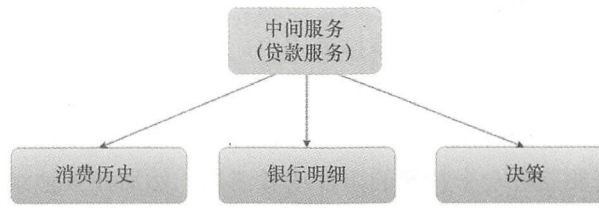
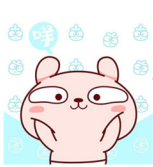


图 3-2

运行和维护这样一个调度服务是有很高成本的。另外一个问题是，这种调度服务自然而然就成了整个架构中单点故障的隐患。当然单点故障也可以使用一些更好的工具和基于云的方法进行化解，如自动扩容、集群等。此外在一个复杂的领域中，编制模式的实现是很有难度的。对于这样的调度控制器，是很难做好分布式设计的。

3.2.2 编排

与编制模式不同的是，编排模式中，每个组件按照各自预先定义的任务来相互协作。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

我们可以将其类比成一个舞蹈团，每个人按照预先定义好的步骤跳舞即可，某些时候舞者需要相互配合，某些时候大家都是同步的（如图 3-3 所示）。



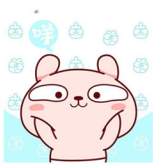
图 3-3

在编制模式中，由调度服务负责调遣所有的服务，而在编排模式中，各个微服务都是相互协同工作的，在交互中会涉及每一个微服务。

在编排模式中，每个服务在做完其工作之后与其他服务进行交互，某些时候是在其自身的工作做到一半时需要触发或初始化下一个服务的工作。该模式中的沟通方式可以是一对一的也可以是一对多或者多对多的。与编制方式类似的是，编排方式可以是同步的也可以是异步的。理想情况下，在编排模式中，各个服务之间会使用一种通用协议进行通信。编排模式中倾向于使用异步通信方式，因为没有中心调度服务的存在，每个组件只需要在完成各自任务的时候广播事件即可，此种方式下各个服务之间实现了解耦和独立。每个组件只需要监听一个特殊的事件或者消息，然后在其他组件发出这种事件或者消息时做出响应即可。

在编排模式中，服务之间通过消息、订阅渠道或者主题来观察其系统环境。在基于消息的通信中，编排模式会非常有用。使用编排方式时，对微服务进行解耦以及添加新服务和移除老服务都非常简单。此种方式可以将系统从编制模式中可能出现的依赖泥潭中解救出来。毫无疑问，编排模式增加了系统的复杂性，每个服务都在运行时产生和消费消息。因此，没有太好的办法能识别事务的确切状态。服务需要有其自己的专门实现才能观察到整个系统的当前状态。这就是这种模式的一些弊端。

在这里的两种模式中，没有哪一种是明显优于另外一种的。在选择的时候要具体问题具体分析。而且在某些系统中，这两种方式都存在。例如，分析大数据的事件流可以



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

使用编排模式，系统内部的主业务可以使用编制模式。下面我们会从另外一个角度来看一下微服务中这些不同的模式。

从另外的一个角度来看，微服务之间的数据流和通信的方式也可以分为同步和异步两种。

3.3 同步通信和异步通信

前一节讨论了微服务间通信是如何触发的。在编制模式中，有一个服务控制所有其他微服务，在编排模式中，一个微服务在其需要与其他微服务通信时发出声音即可。不管最终选择编制模式还是编排模式，总有另外一个问题需要回答：通信过程是同步的还是异步的？也可以用另外一种方式来看待通信是如何完成的：同步方式是一个微服务需要通过提供条件来查询另外一个微服务的数据，并且等待返回结果；异步方式则只需要广播一条消息，告诉所有其他微服务当前我的任务已经完成，后续工作请从我完成的点开始执行。下面我们会继续探讨这两种通信方式。

3.3.1 同步通信

顾名思义，同步通信需要及时的响应。消费者服务器会等待或者阻塞直到接收到远程被调用服务器的响应为止。在同步通信中，接收方当时就知道本次调用是成功还是失败，也可以根据调用状态做出相应的决策。同步通信实现起来较为简单。基于同步通信的这种请求/响应的架构，最佳选择就是 REST 协议。虽然微服务开发人员将基于 HTTP 和 JSON 格式的通信方式视为“一等公民”，但是 SOAP、XMLRPC 以及 socket 编程等方案也可以用来做同步通信。为了加深理解，下面我们以一个库存管理系统作为例子来进行讲解。

假设太阳镜的网站 http://myeyewear.com/的墨镜非常流行。再假设该网站要在圣诞节期间搞促销活动，活动期间会有很多用户会将同一个商品抢到自己的购物车中。在此种情况下，该公司需要在用户添加到购物车的时候检查该商品是否能被添加到购物车。公司可能还想把这个商品的库存冻结起来直到用户成功购买或者时间过期为止。处理这种情况有多种方式，这里为了示例的需要，假设微服务之间是以 REST 的方式通信的，那么整个过程应该由下面这几步组成，如图 3-4 所示。

（1）客户端/消费者来到 API 网关来添加产品到其购物车中。

（2）API 网关带着商品 ID 来到购物车管理服务将商品添加到该用户的购物车。

（3）购物车管理服务通知库存服务冻结给定产品 ID 的产品的库存。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

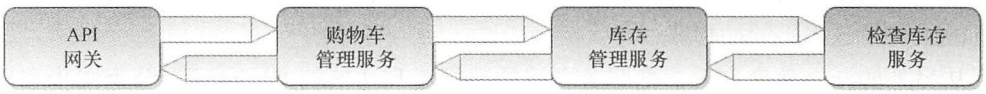


图 3-4

（4）此时购物车管理服务会持续等待库存服务的响应，与此同时库存服务会检查该商品是否可以购买，如果在库存中找到有该商品，那么它会冻结其中一个然后返回给购物车管理服务。

（5）基于收到的返回值，购物车管理服务将该商品添加到用户的购物车中，然后将添加信息返回给其消费者。

在这次通信过程中，购物车服务需要知道库存服务的存在，因此这二者之间其实是有一些依赖关系的。与此同时在等待库存服务响应期间，购物车服务的请求是阻塞的。这其实是一个紧耦合的例子。此时调用方的服务需要处理被调用服务可能出现的多种类型的错误，或许整个被调用服务都挂掉。尽管我们可以用超时来处理这种特殊情况，但是仍然浪费了一些系统资源。而且某些情况下，即便最后调用的那几个服务没有从被调用服务那里得到任何结果，但整个请求还是要等到超时才能返回，这样不仅浪费了系统资源，而且也延长了响应时间。

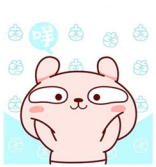
为了处理这种调用服务时出错或者没有响应的情况，可以引入一种断路器模式。在之前的章节中已经简单介绍过断路器，现在让我们来深入探讨一下。

断路器模式会根据某些定义好的规则和阈值来识别错误。然后断路器会介入，在一段时间内阻止继续调用这个有问题的服务，或者启用一些备用方法来计算返回值。在断路器模式中，线路或调用有 3 种状态：

- 关闭状态；
- 开启状态；
- 半开启状态。

断路器在关闭状态时，调用执行正常，调用服务从被调用服务处得到正常结果。断路器在开启状态时，表示被调用服务无法响应，错误数量超过了阈值。断路器开启时，在一个配置的断路时间内，后续请求不再发往被调用服务方，而是直接执行备用方法。断路器最后一种状态是半开启状态，在断路器开启超过了配置时间后，进入这种半开启状态。半开启的时候，请求偶尔会被发往被调用服务，用来检查被调用服务是否已经修复好了，如果线路已经正常，断路器随后会切换为关闭状态，否则就继续恢复到开启状态。

要实现断路器模式，需要在调用方服务中使用一些拦截器。这个拦截器会随时注意发出去的请求和收到的响应。如果有请求得到的失败响应超过了预定义的阈值，拦截器



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

就终止发送后续请求，直接用预先定义的响应或方法来响应，同时会启动定时器来确保一段时间内断路器处于开启状态。

Spring 中的 Netflix Hystrix 就是作断路器的。只需要添加几个注解就可以用这个现成的断路器了。假设有一个在线预定电影票的网站，如果某一个电影订票失败或者超时（如负载太高）后，会推荐另外一个同一时间的场次给用户以便用户能订票成功。

基于以上需求，我们需要创建一个订票应用，和一个调用应用。调用应用会访问订票应用公开的 URL `http://<应用 IP：端口号>/bookingapplication/{userId}/{movieId}`来完成订票。如果这个 URL 访问失败，可以通过一些简单的 Hystrix 注解配置一个备用方法，这其实还需要不少代码：

```
package com.practicalMicroservice.booking;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@SpringBootApplication
@EnableCircuitBreaker

public class ConsumingApplication {

    @Autowired
    private ConsumingService consumingService;

    @RequestMapping(method = RequestMethod.POST,value =
      "/book/{movieId}",produces = "application/json")
          public String book ticket(@PathVariable("userId") String
      userId,@PathVariable("movieId") String movieId) {
          return  consumingService.bookAndRespond(userId,movieId);
    }

    public static void main(String[] args) {
        SpringApplication.run( ConsumingApplication .class, args);
    }
}
```

这里的`@EnableCircuitBreaker`启用了 Spring Boot 应用中的断路器模式。为了在应用中使用 Hystrix 作为断路器模式，还需要另外一个注解`@HystrixCommand`。但是这个注解只能在`@service`或者`@component`注解出现的类中使用。因此，接下来需要



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

创建一个服务类。

```
@Service
public class ConsumingService {

    @Autowired
    private  RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getAnotherCurentlyShowingMovie")
    public String bookAndRespond() {
    URI uri = URI.create("http://<applicationIp:port>/bookingapplication/
    {userId}/{movieId}");

        return this.restTemplate.getForObject(uri, String.class);
    }

    public String getAnotherCurentlyShwoingMovie() {
        return "We are experiencing heavy load on booking of your movie. There
        are some other movies are running on same time, please check if you are
        interested in any one. " + getSameShowtimeMovieList() ;
    }

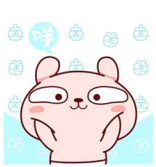
    public String getSameShowtimeMovieList() {
        return "fast and Furious 8, The Wolverine3";
    }
}
```

在这个服务类中，只要线路出现问题，断路器介入之后，它就不会继续调用有问题的服务了，取而代之的是，会开始调用另外一个名为 getAnotherCurrentlyShowingMovie 的方法来告诉用户同一时间上映的其他精彩电影。

断路器非常适合于处理同步通信中的失败场景，但是它解决不了同步通信中的其他问题。知道其他服务的存在加大了服务之间的耦合，而如果耦合太强，微服务其实就跟单体应用没有本质差别了。

3.3.2　异步通信

与同步方式迥异的是，异步通信方式中无须一直等待被调用服务的响应。在这种通信模式中，调用方服务发布请求之后，可以继续完成其后续工作。类似的机制下，调用方服务也有自己的监听器，用来监听其他服务发给该服务的异步请求。监听器会抓取到这些请求并进行处理。异步模式的好处是无阻塞，也不用等待响应。一般而言，这种通信模式会用到消息代理。调用方服务只关心自己的消息是否正确发出了，然后就能继续



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

后续工作了。这种方式下，即使被调用服务根本没有启动也不会影响调用服务。异步通信方式下，各个微服务实现了高度解耦，每个服务不用知道其他服务是否存在。

这种工作模式有一个假设的前提是，在单个服务内处理请求时，一般不需要来自其他服务的数据。特殊情况下，如果这个服务确实需要一些其他服务的数据，那么主要服务在完成自身部分的处理逻辑后，会通过消息通知其他相关的连带服务，如果连带服务也需要这部分主要服务的数据，可以在收到消息后在其自己的上下文中使这部分数据冗余。这样一来在某些时刻，主要服务与连带服务中的数据会出现不一致，但最终数据是能达到一致的。

最终一致性是一个用于在分布式计算中获得高可用的一致性模型。最终一致性可以非正式确保一旦某个数据没有新的更新操作，最终的所有对于该数据的访问都会获得该数据最终修改之后的值。

1. 基于消息/事件的异步通信

异步通信的主要概念是消息从一个点流向另外一个点的过程。这些消息可以是一些命令、事件或者驱动信息等。典型的异步通信有两种类型：

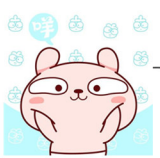
- 基于消息的异步通信；
- 基于事件的异步通信。

这两种类型的工作方式较为类似。某个被调用的服务在完成其自身部分工作之后会触发一条消息，通知其他服务开始后续工作。这两种类型的区别定义得不是很清晰，但是一般而言基于消息的通信是点对点通信的，而基于事件的通信通常是一种发布/订阅模式。在消息驱动型通信中，聚合是通过预定义的消息队列或者通道来完成的。而在事件驱动或者基于命令的通信方式中，预先定义的格式应该是所有服务共享的。消息驱动模式如图 3-5 所示，服务 A 向预定义的队列发布了一条消息，也就是说，逻辑上来讲服务 A 此时是知道服务 B 的存在的，服务 A 知道服务 B 会监听这个队列中的消息然后根据消息内容进行后续动作。



图 3-5

而在事件驱动的通信方式中，服务 A 发出一个事件之后，就可以忘记该事件了。这个发布出来的事件可能会被服务 B 使用，也可能被服务 D 使用，还有可能会被一个最近才刚启动的新服务使用，甚至某些时候可能没有任何服务会读取这个事件，也就是说事件驱动模式与基于消息的通信模式是略有些不同的。事件驱动的通信方式如图 3-6 所示，



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

服务 A 完成其工作之后只管发布事件，而不用关心是否有其他服务会用到这个事件。在消息驱动模式中服务 A 是知道服务 B 一定会监听消息的，因此服务 A 放到消息中的数据应该只有服务 B 能够解析，或者这部分数据只对 B 有用。而在事件驱动方式中，完成的工作应该有一个统一的事件和数据格式，有了这种统一的格式才能够让任何一个服务或者后面新加的服务正确理解和使用。因此，事件驱动的通信方式中，很重要的一点是消息或事件的格式应该对所有服务都是一样的统一格式。图 3-6 描述了事件驱动的这个场景。

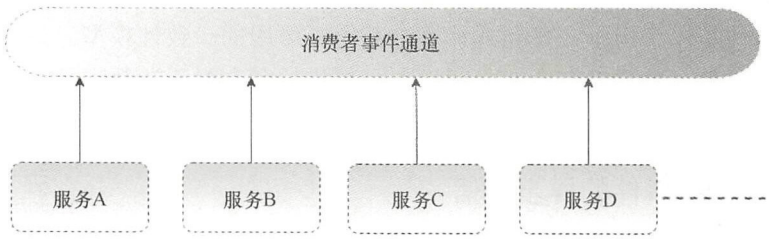


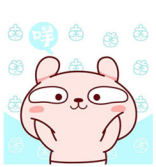
图 3-6

由于这两种方式区别不是很大，从现在开始我们将这两种异步通信方式并称为消息驱动的通信方式。总的看来，消息驱动的异步通信方式有下面这些优点。

- 因为使用异步通信方式使服务之间实现了高度解耦，所以第一个显而易见的好处是各个服务的扩展会更加灵活自如。新开发出来的微服务可以悄无声息地加入整个架构中来接收一些消息/事件，然后有针对性地做出一些行为。消息的发起方根本就不用关心接收消息的到底有哪些微服务。
- 在消息驱动型通信方式中，回压可以用一种比同步通信更好的方式得到解决。如果消息的产生方生成消息过快，超过了消费者的消费速度，就会有大量的消息堵塞在队列中等候处理。这种情形就是所谓的回压。在出现回压的时候，消费者可以做两件事：一是通过发布一个特殊事件来告诉生产方减缓速度，二是静静地添加更多的工作单元来加快消费速度。这类问题通常会出现在事件处理过程中。

当然异步通信方式也有其弊端。

- 运行的复杂性会增加运维成本，因为需要额外安装、配置和维护高可用的消息代理。
- 观察系统中某个时间点的准确状态变得更加困难，因为消息流和事件会分发到很多不同的服务中去，因为微服务的自治性，所以想要了解清楚整个系统就更加困难。因此，在出现问题进行调试的时候也会需要更多的成本。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- 某些时候，将原本请求/响应结构转换成基于消息的模式是困难的。以一个火车票订票系统为例，如果不在订票之前检查有没有空的座位，那么是无法正确受理订票请求的。也就是说这种有明显先后顺序的地方，使用同步方式更合理。

2. 使用 REST 实现异步通信方式

异步通信方式既可以用 REST 风格的方式来实现，也可以用消息代理的方式来实现。如果用 REST 风格来做的话，则需要一个作为消息处理中心的微服务。每个微服务将消息推送到这个处理中心，然后消息中心复杂将消息分发给相应的接收方微服务。在这种方式中，发起方的服务需要等待确认消息提交成功，但是并不需要等待被调用方的微服务处理完整个业务之后的响应。其工作原理如图 3-7 所示。

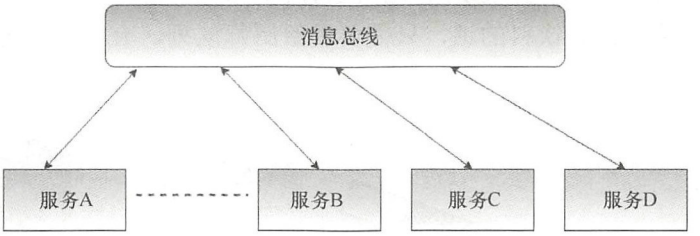


图 3-7

当服务 A 完成其工作之后，提交给消息中心一个完成消息，消息中心有两种方式来将该消息发给被调用的其他服务。第一种方式是消息中心智能识别出该消息的接收方应该是哪个服务，然后将消息和数据准确地发送给该服务。另外一种方式是，消息中心直接将收到的消息分发给每一个在消息中心上注册的微服务，由每个微服务自己判断是应该忽略该消息还是采取后续动作。

3. 使用消息代理的方式实现异步通信

为了讲解消息代理的异步通信实现方式，让我们以一个名为“Crispy Bun”的汉堡店作为例子。这是一个得来速式的汉堡店，顾客可以在一个窗口下好订单之后，在下一个窗口等着订单，中途一直不用下车。这个汉堡店的订单系统可以在第一个窗口处接收订单，然后将订单信息封装成消息或者事件之后，推送到某个队列或者主题。在每个厨师面前有一个组件用于展示所有用户的订单。我们的需求是订单提交到某个中间代理（队列或主题）之后，所有厨师都可以监听这些新来的消息。即便厨师都在忙，订单也不会从列表中消失，如果某个厨师看完订单信息开始准备订单内容的时候，就将订单保存



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

下来，从代办列表中撤离。这个例子还可以展开很多细节，例如同一主题最多的订阅数，再如订单消息可以同时发给厨师和打包团队，在厨师准备食物的时候，打包团队可以准备些盒子用于装汉堡，又或者如果订单中有饮料，还需要同时发给饮料部门，同时准备配制饮料。所以不同组件可以同时监听订单消息，然后各自采取相应的行动。简单起见，只考虑只有一个接订单组件和一个厨师组件的简单示例。

RabbitMQ 是目前最为流行的实现了**高级消息队列协议**（Advanced Message Queuing Protocal，AMQP）协议的消息代理解决方案之一。它是基于 Erlang 语言的，跟普通消息代理直接将消息发布到队列中不同的是，RabbitMQ 中需要先经过一个消息交换机。交换机将消息按照属性、连接和路由队列分发路由给不同队列的组件。一个交换机上可以有多个不同的队列，通过不同的路由键进行区分。例如，在我们的例子中，订单中可能会有饮料，然后相应地我们需要两种绑定键来区分，即绑定 Chef_key 和绑定 All_key。其中绑定 All_key 会发给厨师和饮料两个队列，而绑定 Chef_key 则只发给厨师队列。如果订单中没有饮料的话，只需要发给厨师队列，会使用绑定 Chef_key，如图 3-8 所示。

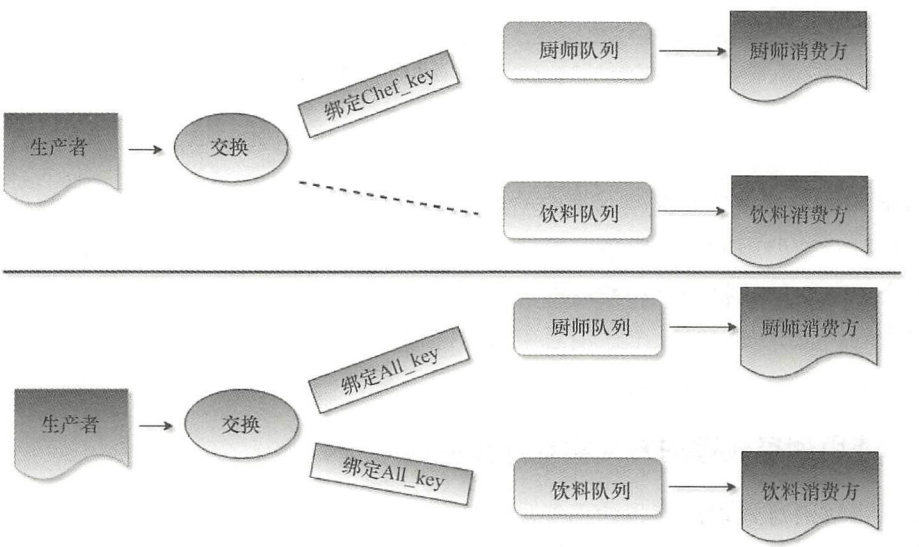
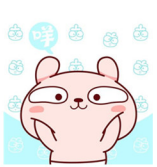


图 3-8

如果对 RabbitMQ 不熟悉，强烈建议先学习一些与 RabbitMQ 相关的知识。下面是在 Unix 型操作系统的机器上安装 RabbitMQ 的步骤的简单描述。

（1）执行 `sudo apt-get update` 命令。

（2）使用下面的命令添加 RabbitMQ 应用代码库：



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
echo "deb http://www.rabbitmq.com/debian/ testing main" >>
/etc/apt/sources.list
curl http://www.rabbitmq.com/rabbitmq-signing-key-public.asc |
sudo apt-key add
```

（3）使用 sudo apt-get update 命令再次更新。

（4）运行 sudo apt-get install rabbitmq-server 命令来安装 RabbitMQ。

执行完上述这些操作之后，会在机器上安装 Erlang 和 RabbitMQ，并且启动 RabbitMQ 服务。如果服务没有自动启动，也可以使用 service rabbitmq-server start 命令来手动启动 RabbitMQ。安装完这些之后，可能需要一个拥有管理员权限的控制台来处理 RabbitMQ 的集群和节点。需要通过以下命令安装 RabbitMQ 插件：

```
sudo rabbotmq-plugins enable rabbitmq_management
```

执行完上述命令之后，就可以在浏览器中打开 http://<你的本机 ip 或者 localhost>:15672 来查看 RabbitMQ 的管理控制台，大概如图 3-9 所示。打开管理控制台需要用户名和密码，默认都是 guest。

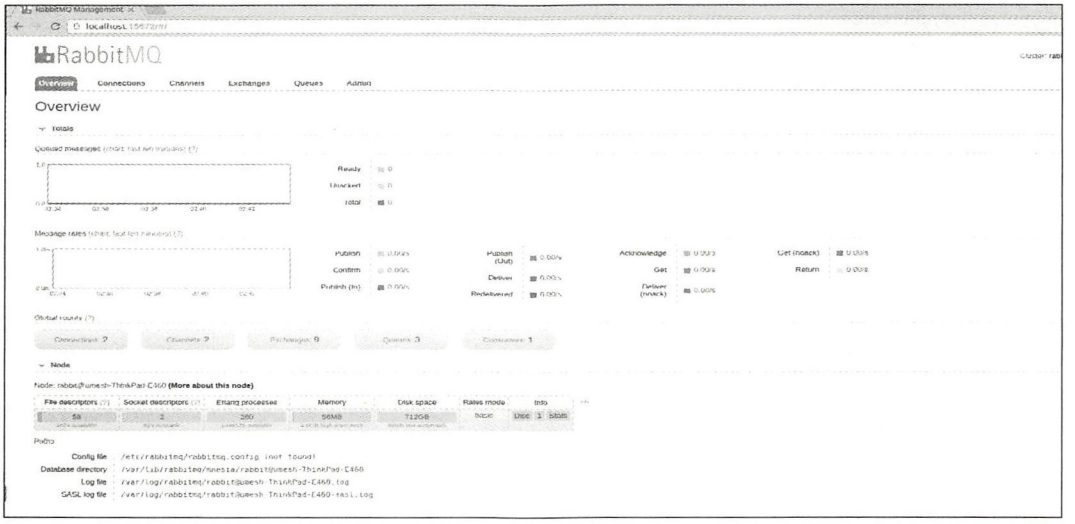


图 3-9

前面已经提过，对这个示例而言需要两个应用，一个是消息的生产者，另一个是消费者。目前 RabbitMQ 已经启动，还需要给生产者写一点代码，让它能真正产生订单，并且将订单消息提交到默认的交换机中去。默认的交换机没有名字，是一个空字符串""，此时消息会被转发到名为 crispyBunOrder 的队列中去。

订单生产者的 pom.xml 文件如下：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.practicalMircorservice</groupId>
    <artifactId>EventProducer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>EventProducer</name>
    <description>com.practicalMircorservice </description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
        </project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                <version>Camden.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

订单生产者应用中，还需要有下面这两个 Java 文件：

- EventProducerApplication.java：这个文件是应用程序的主应用类，我们会将 REST 控制器直接放在主应用类中。
- CrispyBunOrder.java：这个是用来定义订单对象的类。

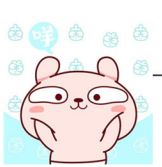
EventProducerApplication.java 的代码如下：

```
@SpringBootApplication
@EnableBinding
@RestController
public class EventProducerApplication {

    private final String Queue = "crispyBunOrder";
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public static void main(String[] args) {
        SpringApplication.run(EventProducerApplication.class, args);
    }

    @RequestMapping(method = RequestMethod.POST, value = "/orders/{orderId}")
    public void placeOrder(@PathVariable("orderId") UUID orderId, @RequestParam
    ("itemId") Integer
    itemId,@RequestParam("userName") String userName) {
        CrispyBunOrder orderObject = createOrder(orderId,itemId,userName);
        rabbitTemplate.convertAndSend(Queue,orderObject);
    }
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    private CrispyBunOrder createOrder(UUID orderId,Integer itemId,
    String userName){
        CrispyBunOrder order = new CrispyBunOrder();
        order.setItemId(itemId);
        order.setOrderId(orderId);
        order.setUserName(userName);
        order.setOrderPlacedTime(new Date());
        return order;
    }
}
```

这个类的主要业务方法中，会接受一组 orderId、itemId 和 userName 参数，然后创建一个订单，接着将该订单提交到名为 crispyBunOrder 的队列中。因为我们已经在前面的**项目对象模型**（Project Object Model，POM）文件中添加了与 RabbitMQ 相关的依赖，Spring Boot 会自动创建一个 RabbitMQ 模板。有了 RabbitMQ 模板，就可以将任何对象发送到给定的队列名中去。这里其实是发给默认交换机的，默认交换机没有名字，只是一个空字符串""。因此，任何发往默认交换机的消息都会直接定向到相应名字的队列。

CrispyBunOrder 类有 4 个属性，其内容如下：

```
package com.practicalMircorservices.eventProducer;

import java.io.Serializable;
import java.util.Date;
import java.util.UUID;

public class CrispyBunOrder implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = 6572547218488352566L;

    private UUID orderId;
    private Integer itemId;
    private Date orderPlacedTime;
    private String userName;
    public UUID getOrderId() {
        return orderId;
    }
    public void setOrderId(UUID orderId) {
        this.orderId = orderId;
    }
    public Integer getItemId() {
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        return itemId;
    }
    public void setItemId(Integer itemId) {
        this.itemId = itemId;
    }
    public Date getOrderPlacedTime() {
        return orderPlacedTime;
    }
    public void setOrderPlacedTime(Date orderPlacedTime) {
        this.orderPlacedTime = orderPlacedTime;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

该应用的 application.properties 配置文件中有如下两个属性：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

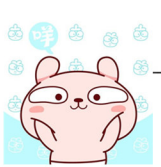
5672 是 RabbitMQ 的默认端口。

现在轮到了消费者一方了。消费者是另外一个名为 EventConsumerApplication 全新的应用。这个应用可以直接从 start.sprig.io 上生成之后下载。下载之前要确保勾选了 **Stream Rabbit** 复选框。在这个应用中，需要有一个与生产者一模一样的 CrispyBunOrder 类，因为这里需要对这个类的数据进行反序列化。除此之外，还应该有一个监听器，用于监听 RabbitMQ 中名为 crispyBunOrder 的队列。这个类的代码如下：

```
package com.practicalMircorservices.eventProducer;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.messaging.handler.annotation.Payload;


@SpringBootApplication
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@RabbitListener(queues = "crispyBunOrder")
@EnableAutoConfiguration
public class EventConsumerApplication {

    @Bean
    public Queue crispyBunOrderQueue() {
        return new Queue("crispyBunOrder");
    }
    @RabbitHandler
    public void process(@Payload CrispyBunOrder order) {
        StringBuffer SB = new StringBuffer();
        SB.append("New Order Received : \n");
        SB.append("OrderId : " + order.getOrderId());
        SB.append("\nItemId : " + order.getItemId());
        SB.append("\nUserName : " + order.getUserName());
        SB.append("\nDate : " + order.getOrderPlacedTime());
        System.out.println(SB.toString());
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(EventConsumerApplication.class, args);
    }

}
```

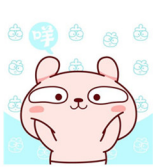
这里我们通过@RabbotListener(queues = "crispyBunOrder")注解定义了应该监听的队列名称。除此之外，我们还可以在这里定义多个其他参数，如交换机名字、路由键等。例如，下面这样的注解：

```
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(value = "myQueue", durable = "true"),
        exchange = @Exchange(value = "auto.exch"),
        key = "orderRoutingKey")
)
```

在不同控制台中使用 mvn spring-boot:run 分别启动这两个应用/组件。确保应用的 application.properties 文件中的 server.port 属性使用的是两个不同的端口，否则会有端口冲突。

现在可以在命令行中使用 curl 命令访问产生订单应用的 URL 来测试返回值。例如，下面这样一个命令：

```
curl -H "Content-Type: application/x-www-form-urlencoded" --data "itemId=
1&userName=john"
http://localhost:8080/orders/02b425c0-da2b-445d-8726-3cf4dcf4326d;
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在执行完这个命令后，就能在消费者的控制台中看到订单信息了。通过这个例子可以对基于消息的异步通信方式有个简单的理解。在实际生活中有很多复杂的例子，例如，从网站获得用户操作、股票价格的变动等。大部分时候使用消息代理都比较方便，例如，对于一个输入的数据流，需要基于路由的键来将数据分发给不同队列等类似场景。Kafka是另外一个不错的解决这种问题的工具。Spring 提供了对 Kafka、RabbitMQ 和一些其他消息代理的内置支持。有了 Spring 的内置支持，开发人员可以快速搭建和开发一个基于消息代理的应用。

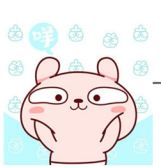
3.3.3 财务服务

继续扩展我们的信用评分应用，我们需要添加一个名为 FinancialService 的微服务。在第 2 章中，我们创建了一个 UserService 微服务，用于存储用户的个人信息和地址。在本章中，我们会添加这个财务微服务，用来保存用户的财务详情信息，包括银行账号详情和用户的财务负债情况等。这个微服务的表结构如下：

```
CREATE TABLE `bank_account_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `bank_name` varchar(100) DEFAULT NULL,
  `account_number` varchar(20) NOT NULL,
  `account_holders_name` varchar(250) NOT NULL,
  `fsc` varchar(20) DEFAULT NULL,
  `account_type` int(11) DEFAULT NULL,
  `created_on` datetime(6) NOT NULL,
  `deleted_on` datetime(6) DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;

CREATE TABLE `obligations_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `total_monthly_spending` decimal(10,0) NOT NULL,
  `monthly_income_supports` decimal(10,0) NOT NULL,
  `monthly_emi_payout` decimal(10,0) NOT NULL,
  `created_on` datetime(6) NOT NULL,
  `deleted_on` datetime(6) DEFAULT NULL,
  PRIMARY KEY (`Id`),
  KEY `Obligations_user_id` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Bootstrap 属性文件中将包含下面这样两个属性：



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
spring.application.name=financialService
spring.cloud.config.uri=http://localhost:8888
```

相应地，我们还需要在 configServer 代码中创建 financialService.properties 文件给财务服务使用的：

```
server.port=8090
Connect.database=financialDetails
spring.datasource.url=jdbc:mysql://localhost:3306/financialDetails
spring.datasource.username=xxxxxx
spring.datasource.password=xxxxxxxx
# optional Properties
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.naming_strategy = org.hibernate.cfg.ImprovedNamingStrategy
```

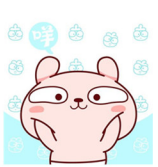
这个应用的创建过程与 UserService 应用的差不多。需要从 start.spring.io 下载模板代码。应用的名字可以就叫 FinancialServiceApplication。这个服务的主应用文件跟用户服务应用的也差不多是一样的：

```
package com.practicalMicroservcies;

import org.flywaydb.core.Flyway;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class FinancialServiceApplication {
    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String userName;
    @Value("${spring.datasource.password}")
    private String password;
    @Value("${Connect.database}")
    private String database;

    @Bean(initMethod = "migrate")
    public Flyway flyway() {
        String urlWithoutDatabaseName = url.substring(0,url.lastIndexOf("/"));
        Flyway flyway = new Flyway();
        flyway.setDataSource(urlWithoutDatabaseName, userName, password);
        flyway.setSchemas(database);
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        flyway.setBaselineOnMigrate(true);
        return flyway;
    }

    public static void main(String[] args) {
        SpringApplication.run(FinancialServiceApplication.class, args);
    }
}
```

因为这个服务有两张表，相应地我们需要两个实体表，即银行账号详情和债务详情。这些文件里的代码如下：

```
@Entity
@Table(name = "bank_account_details")
public class BankAccountDetail implements Serializable {

    private static final long serialVersionUID = 4804278804874617196L;

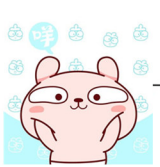

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;

    /**
     * User Id , unique for every user
     */
    @Column(name = "user_id", nullable = false, unique = true, length = 36)
    private String userId;

    /**
     * Bank Name of User
     */
    @Column(name = "bank_name", nullable = false, length = 100)
    private String bankName;

    /**
     * Account number of User
     */
    @Column(name = "account_number", nullable = false, unique = true,
    length = 20)
    private String accountNumber;

    /**
     * Account holder Name of account
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
     */
    @Column(name = "account_holders_name", nullable = false, length = 250)
    private String accountHolderName;

    /**
     * Any National Financial system code of bank
     */
    @Column(name = "fsc", nullable = false, length = 20)
    private String fsc;

    /**
     * Account Type of user
     */
    @Column(name = "account_type", nullable = false)
    private Integer accountType;

    /**
     * Date on which Entry is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP")
    private Date createdOn = new Date();

    /**
     * Date on which Entry is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT NULL")
    private Date deletedOn;
}
```

下一个实体类是债务实体，用于存放用户的债务信息，代码如下：

```
@Entity
@Table(name = "obligations_details")
public class ObligationDetails implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -7123033147694699766L;

    /**
     * unique Id for each record.
     */
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;

    /**
     * User Id , unique for every user
     */

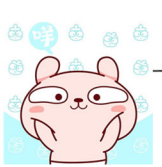
    @Column(name = "user_id", nullable = false, unique = true, length = 36)
    private String userId;

    /**
     * Totally monthly income , may be from himself or wife (combined)
     */
    @Column(name = "monthly_income_supports", nullable = false)
    private BigDecimal monthlyIncome;

    /**
     * Monthly Emi to payout
     */
    @Column(name = "monthly_emi_payout", nullable = false)
    private BigDecimal monthlyemi;
    /**
     * totally month spending
     */
    @Column(name = "total_monthly_spending", nullable = false)
    private BigDecimal monthlySpending;

    /**
     * Date on which Entry is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP")
    private Date createdOn = new Date();

    /**
     * Date on which Entry is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT NULL")
    private Date deletedOn;
}
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这个类中还应该有相应的 setter 和 getter 方法。

当然，还可以重载 toString 方法：

```
@Override
public String toString() {
    return "ObligationDetails [userId=" + userId + ",
    monthlyIncome=" + monthlyIncome + ", monthlyemi=" + monthlyemi
    + ", monthlySpending=" + monthlySpending + ", createdOn=" +
    createdOn + ", deletedOn=" + deletedOn + "]";
}
```

在用户服务中会有与上面提到的两个实体（银行账号详情和债务详情）对应的代码库类：

```
package com.practicalMicroservcies.repo;
import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.BankAccountDetail;

public interface BankAccountDetailRepository extends
JpaRepository<BankAccountDetail, Integer> {
    BankAccountDetail findByUserId(String userId);
}
```

债务实体相关的代码库：

```
package com.practicalMicroservcies.repo;

import org.springframework.data.jpa.repository.JpaRepository;

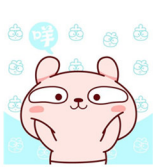
import com.practicalMicroservcies.entity.BankAccountDetail;
import com.practicalMicroservcies.entity.ObligationDetails;

public interface ObligationRepository extends
JpaRepository<ObligationDetails, Integer> {
    ObligationDetails findByUserId(String userId);
}
```

还有一个服务类名为 FinancialServices，用于处理来自两个代码库的一些查询：

```
@Service
@Transactional
public class FinancialServices {

    @Resource
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
BankAccountDetailRepository accountDetailRepo;

@Resource
ObligationRepository obligationRepo;
```

下面的方法是用来保存用户的银行账号详情的：

```
public void saveAccountDetail(BankAccountDetail accountDetail) {
    accountDetailRepo.save(accountDetail);
    System.out.println("AccountDetails Saved!");

}

public void saveObligation(ObligationDetails obligationDetails) {
    obligationRepo.save(obligationDetails);
    System.out.println("Obligation Details Saved!");

}
```

还需要方法根据提供的 userId 来从数据库提取银行账号详情和债务详情。下面是该服务类中的两个方法，用来完成获取数据的工作：

```
public ObligationDetails getObligationDetail(UUID userId) {
    ObligationDetails returnAddressObject =
    obligationRepo.findByUserId(userId.toString());
    return returnAddressObject;

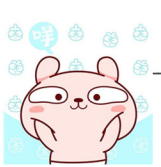
}

public BankAccountDetail getAccountDetail(UUID userId) {
    BankAccountDetail userObjectToRetrun =
    accountDetailRepo.findByUserId(userId.toString());
    return userObjectToRetrun;

}
```

下面是删除用户财务详情的方法。这个方法并不会真的从数据库里删除数据，而只是将数据标记为已删除。事实上，对于数据库，所有彻底删除数据的语句都应该尽量避免：

```
public void deleteFinancialDetail(UUID userId) {
    BankAccountDetail accountObject =
    accountDetailRepo.findByUserId(userId.toString());
    accountObject.setDeletedOn(new Date());
    accountDetailRepo.saveAndFlush(accountObject);
    ObligationDetails obligationObject =
    obligationRepo.findByUserId(userId.toString());
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            obligationObject.setDeletedOn(new Date());
            obligationRepo.saveAndFlush(obligationObject);
        }

    }
```

控制器如下所示：

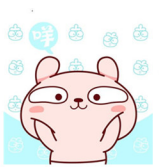
```
@RestController
@RequestMapping("/PM/finance/")
public class FinancialController {
    private static final Logger logger =
Logger.getLogger(FinancialController.class);

    @Resource
    FinancialServices financialService;

    @Resource
    ObjectMapper mapper;
    /**
     * Method is responsible for adding new AccountDetails.
     *
     * @param address
     * @param userId
     * @return
     */
    public static final String addAccountDetails =
    "addAccountDetails(): ";

    @RequestMapping(method = RequestMethod.POST, value = "
    {userId}/account", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<String> addAccountDetails(@RequestBody
    BankAccountDetail accountDetail, @PathVariable("userId") UUID
    userId) {
        logger.debug(addAccountDetails + " Account for user Id " +
        userId + " is creating.");
        accountDetail.setUserId(userId.toString());
        financialService.saveAccountDetail(accountDetail);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    /**
     * Method is responsible for creating a obligation Details.
     *
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
     * @param userDetail
     * @param userId
     * @return
     */

    public static final String addObligationDetails =
    "addObligationDetails(): ";

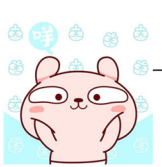
    @RequestMapping(method = RequestMethod.POST, value =
    "{userId}/obligation", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<String> addObligationDetails(@RequestBody
    ObligationDetails obligationDetails, @PathVariable("userId") UUID
    userId) {
        logger.debug(addObligationDetails + " Creating user's
        obligation with Id " + userId + " and details : " + obligationDetails);
        obligationDetails.setUserId(userId.toString());
        financialService.saveObligation(obligationDetails);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    /**
     *  Deleting Financial Detail of user
     * @param userDetail
     * @param userId
     * @return
     */

    public static final String deleteFinancialDetails =
    "deleteFinancialDetails(): ";

    @RequestMapping(method = RequestMethod.DELETE, value = "{userId}",
    produces = "application/json", consumes = "application/json")
    public ResponseEntity<String> deleteFinancialDetails(
    @PathVariable("userId") UUID userId) {
        logger.debug(deleteFinancialDetails + " deleting user with Id "
        + userId);
        financialService.deleteFinancialDetail(userId);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    /**
     * Method is responsible for getting the account detail for given ID.
     *
```



仅供非商业用途或交流学习使用

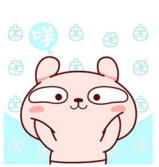
仅供非商业用途或交流学习使用


```
     * @param userId
     * @return
     */
    public static final String getAccountDetails =
    "getAccountDetails(): ";

    @RequestMapping(method = RequestMethod.GET, value =
    "{userId}/account", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<BankAccountDetail>
    getAccountDetails(@PathVariable("userId") UUID userId) {
        logger.debug(getAccountDetails + " getting information for userId " +
        userId);
        BankAccountDetail objectToReturn =
        financialService.getAccountDetail(userId);
        if (objectToReturn == null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
    }

    /**
     * Method is responsible getting the Obligation Detail.
     * @param userId
     * @return
     */
    public static final String getObligationDetails =
    "getObligationDetails(): ";

    @RequestMapping(method = RequestMethod.GET, value = "
    {userId}/obligation", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<ObligationDetails>
    getObligationDetails(@PathVariable("userId") UUID userId) {
        logger.debug(getObligationDetails + " getting Obligation
        Details for user Id: " + userId);
        ObligationDetails objectToReturn =
        financialService.getObligationDetail(userId);
        if (objectToReturn == null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
    }
}
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

至此，我们就完成了财务微服务的所有代码。该服务的端点 URL 如下所示。

对于创建债务，端点 URL 是：

```
POST http://localhost:8090/PM/finance/<user Id>/obligation
```

这个 URL 的例子如下：

```
POST
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/obligation
```

对于创建银行账号信息，端点 URL 是：

```
POST http://localhost:8090/PM/finance/<user Id>/account
```

这个 URL 的例子如下：

```
POST
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/account
```

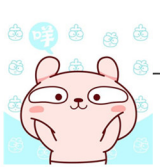
如果对统一 URL 发出 GET 请求，会从数据库中提取用户的详情信息，然后以 JSON 格式发回。例如，如果想要知道任何用户的债务信息，可以通过发送 GET 请求到上面提到的端点 URL 来获取：

```
GET
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/obligation
```

这个 API 的响应是一个类似这样的 JSON 数据：

```
{
   "id": 1,
   "user_id":"3a52ce4-9331-43b5-8b56-09bd62cb0444",
   "monthlyIncome":150000,
   "monthlyemi":50000,
   "monthlySpending":50000,
   "createdOn":"1483209000000",
   "deletedOn":null
}
```

该服务用于存储用户的财务信息。它由两张表组成，一张表用来存储用户的银行详情信息，另一张表用来存储用户的月收入、家庭花销、月分期付款 EMI 花销等。所有这些决定会帮助我们创建信用评分微服务。按照相同的方式，还会有另外一个职业微服务，用于存储职业详情信息，如薪水、是否为个体户、当前雇主是谁以及所有雇员的当前薪水等。这个微服务的实现方式跟现在有的几个微服务非常类似，就留待读者自行实现。

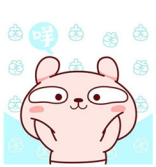


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.4　小结

本章讨论了微服务之间的一些通信原则、各种通信方法（如同步通信和异步通信）以及通信模式（如编制和编排）。同时，还通过一个基于事件的消息队列的例子，讨论了实现基于事件的消息队列的多种场景和不同的方式。REST 可以用在同步和异步两种通信类型中。这两种通信方式都有一些对应的工具。在选择合适的通信方式时最主要的还是要结合使用场景和具体项目的需要。第 4 章将会继续以信用评分微服务为例讲解微服务中的安全问题。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 4 章
微服务端点的安全

上一章主要讨论了微服务之间多种不同的通信方式。总的来说有两种方式，即同步通信模式和异步通信模式。在微服务架构中，经常会有这样一个问题：如何解决微服务间的内部通信以及微服务与外部通信的安全问题？微服务架构中每一个服务的职责除了完成一套业务领域的功能，还隐含了一部分职责是确保通信安全，这其实一定程度上违背了微服务设计的原则。所谓的安全问题主要是两个部分——认证和授权。

本章将会深入探讨微服务中的安全问题。

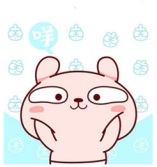
本章将会覆盖下列主题：

- 微服务的安全挑战；
- 使用 JWT（JSON Web Token）以及 OpenID 和 OAuth 2.0；
- 在一个应用中如何实施 JWT 认证方式；
- 信用评分微服务的后续开发。

4.1 微服务的安全挑战

软件开发是一个瞬息万变的行业，随时都会有一些新技术涌入这个行业，与技术本身的发展息息相关的是软件的安全问题。在任何软件架构中，安全都不是一个首要关注的问题，但是又都少不了安全方面的考虑。在单体应用中，暴露到外部世界的地方相对较少，但是只要哪一个地方稍不注意，就会给恶意攻击者攻击整个系统以可乘之机，严重的时候整个系统都会因为这仅有的一个缺口而万劫不复。由于单体应用存在已久，相应的漏洞攻击也随之形成了一些模式。所谓道高一尺魔高一丈，随着应对这些常见攻击模式的经验总结，也逐步开发了一些软件标准用于有效地抵御这些攻击。常见的攻击类型有电子诈骗、恶意篡改、**阻断服务**（denial-of-service，DOS）、升级或增加特权以及信息泄露等。

这些攻击同样有可能发生在微服务架构中。从某种意义上说，微服务相较于单体应用还有一个安全方面的好处是：攻击者即使攻破了一个微服务，其他服务也不会受到影



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

响，因为服务之前有一定的隔离性，而且每个微服务有自己的安全层。虽然微服务架构与单体架构有很多不同，但是就防止微服务被攻击而言，还是有不少策略是可用的。下面我们将分门别类讨论一下微服务环境中所面临的一些挑战和一些安全实践。

4.1.1　复合技术栈或者存在遗留代码

在微服务中，可以有多重不同的技术栈并存（译者注：可以不等于一定会有）。在微服务中有多种不同的技术栈存在的时候，要做一个统一的安全层是需要很多开发成本的，不同语言实现起来有不同的复杂性。而且如果系统中有一部分遗留代码的话，会进一步增加这种复杂性，因为开发人员还得与遗留代码保持一致性。遗憾的是，受遗留代码的限制，通常我们不得不在新的安全层方面做些妥协。

4.1.2　认证和授权（访问控制）

这是安全上下文中的两个重要概念，有时候也统称为认证。认证（authentication）做的事情是明确“你是谁”，授权（authorization）解决的问题则是“你能做什么”。登录到任何一个系统的过程就是认证的过程。在登录之后给用户分配一种角色的过程是授权，例如，管理员和普通用户能看到的数据通常是不同的。针对不同的角色，会有相应的访问控制。每个实体都应该在能满足业务需要的情况下设置尽可能低的访问权限。

4.1.3　基于令牌的安全实践

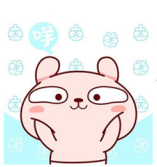
如果开发人员选择的是基于令牌的安全实践，那么紧接着就会面临选择令牌格式的挑战。令牌的管理方式要十分小心，常见的需要考虑的问题有：令牌的生命周期、过期时间，以及是否会存储在某些被人入侵之后做些破坏性的事的设备上。这里的最佳实践是使用 JWT 格式的令牌。

4.1.4　安全性的责任

一般而言，开发人员会习惯性地认为安全应该是网络团队或部署团队的职责。然而在微服务环境中，每个服务都会面临安全方面的威胁。当然网络层有网络层的安全问题和安全策略，如端口锁定、IP 白名单等。但是对于很多组织来说，要说服开发人员转变思维在写代码的时候考虑安全问题并记录一些可疑活动也是个不小的挑战。

4.1.5　编制风格的风险

采用编制模式的微服务也会面临一个威胁，因为在编制模式中，由一个服务组件总



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

控所有其他微服务，所以一旦这个总控微服务被攻破，那么所有组件的行为就都被破坏或被控制了。因此，这种由一个组件控制全局的方式是有安全风险的。

4.1.6 微服务之间的通信

在微服务架构中微服务之间有大量的数据流，预防数据被篡改是另外一个问题，例如，调用服务如何确定收到的数据在通信过程中是未经篡改的。同时，还需要一些标识符来区分敏感数据，敏感数据需要更加高度关注其安全性。常见的敏感数据有支付数据等。

上述讨论只触及了冰山一角，微服务中还有很多别的安全方面的挑战。API 网关也可以用来解决一部分安全问题。幸运的是，我们可以采用一些标准安全实践来有效地规避一些风险和挑战。

在微服务架构中，有这样一些通用的安全实践可以考虑采用。

- 最基础的是通信的证书检查，例如，使用 HTTPS 就是一个不错的开始（如果使用的是基于 REST 的通信方式）。
- 微服务通信时的数据加密。
- 使用正确的访问策略，组件之间的访问应该是受限制的，应该仅提供满足业务所需的最低级别许可。
- 处理安全问题的另外一个方面是监控工具。如果有了很好的监控工具，就可以很容易看出来系统中发生了什么。
- 记录所有所需的日志用于调试和识别可能的安全问题，对于开发团队也是十分有帮助的。
- 定期对安全设施进行测试。
- 如果你的微服务中用到了第三方代码或者容器，需要在使用之前进行充分的验证，而且要随时关注这些第三方的安全补丁。
- **OWASP**（Open Web Application Security Project）提供了一些非常有用的安全建议，在考虑安全问题时也可以参考 OWASP 的建议。
- 团队内需要频繁地做代码评审，并保证每次评审的代码数量较少。如果请人看 1000 行代码，你肯定得不到任何建议，如果是不足 100 行代码你就能得到一些建议了。

一般来说，微服务架构中都会采用一种基于令牌的安全认证系统。这种策略不只适用于基于 REST 的请求，也可以用于其他格式的通信方式。令牌的原理是，客户端首先会获取令牌，然后将令牌信息嵌入所有后续请求中。在后台服务认证和授权之后，会将令牌发送给客户端。资源拥有者会通过令牌生成服务器来验证令牌，以及检查令牌是否



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

有访问当前资源的权限。如果令牌过期了，客户端需要给令牌提供方再次出示证书来获取新令牌。基于 REST 的微服务中最常用的令牌方式是 JWT。JWT 更进了一步，它将认证信息和权限数据以某种加密算法存放在客户端的令牌中。下一节会详细讨论认证和授权架构以及 JWT 令牌的内容。

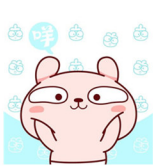
4.2　与 OpenID 的 OAuth 2.0 一起使用 JWT

JWT 是 JSON Web Token 的简写，它具有与特定调用相关的一些信息。JWT 可以同时用于认证和授权。我们可以从一个有效的 JWT 令牌中知道用户是谁以及这个用户能干什么。在了解 JWT 的结构之前，我们先熟悉一下另外两个概念——OpenID 和 OAuth。时至今日 OAuth 2.0 和 OpenID Connect 已经成为**安全声明标记语言**（Security Assertion Markup Language，SAML）的替代品。它们主要是用于用户身份及信息传达给身份提供方以及服务和资源提供方进行用户认证的。OpenID 主要是用于认证的，OAuth 则主要用于授权。下面几节会详细讨论这两个概念。

4.2.1　OpenID

退回到几年前，每个网站都有自己的登录机制。用户需要自己维护每个网站的登录信息。例如，用户在购物、旅行、订票时使用的邮箱、网站都各不相同，因此用户需要记住各个网站的登录信息，这对于用户而言是个困扰。为了解决这个问题，有人提出了**单点登录**（Single Sign-on，SSO）的概念。单点登录可以让用户使用同一个用户名和密码来登录许多不同的网站。如此一来，登录过程变得更加快捷和简单了。也正是在这样的时代背景下，OpenID 登上了历史舞台。OpenID 是一种在用户访问过的所有网站上认证该用户登录的方式。有了 OpenID 之后，用户只需要向 OpenID 提供方提供初始密码，OpenID 提供方就可以告诉网站这个登录的用户是谁。使用这种方式之后，用户访问的网站根本不用保存用户密码。

市场上有很多 OpenID 的提供方，一般都会列出来任由挑选。例如，你经常会看到一些网站的登录页面上有一个“Log in with Google+”的登录链接。使用这个链接的意思是，如果你是谷歌 OpenID 的用户，你可以使用谷歌的 OpenID 来登录这些网站。其实只要你拥有一个 Gmail 或者谷歌账户就意味着你是一个谷歌 OpenID 的用户了。除了谷歌外还有一些非常流行的网站，如 Facebook、Twitter、LinkedIn 等，也都提供了这种 OpenID 的机制（译者注：国内提供方有微博、微信、支付宝等）。当你使用这些 OpenID 登录的时候，会被重定向到提供方的页面来输入相应的用户名和密码，验证之后会再次重定向



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

回来。例如，如果你使用 Gmail 账号登录某个网站，登录请求其实是发给 Google OpenID 服务器的，在 Google 验证了你的登录信息之后，会将登录信息（如登录邮箱等）共享给你最开始访问的网站。在共享用户信息之前，OpenID 服务器需要征得用户同意，用户可以选择同意或者拒绝该网站获取用户信息。

图 4-1 简单解释了一个普通 OpenID 提供方的工作方式。

（1）用户选择使用 OpenID 提供方的账号进行登录（如谷歌）。

（2）**依赖方**（RP）（也就是用户最开始访问的网站）发请求给 OpenID 的提供方来认证用户和共享信息。

（3）OpenID **提供方**（OP）采集用户的登录信息，同时征求用户是否同意共享个人信息给该第三方网站使用。

（4）如果用户选择了允许，登录信息会被发送给 OP，紧接着由 OP 验证用户的登录信息是否正确。

（5）OP 将登录信息和个人信息分享给 RP。

（6）RP 使用 OP 提供的用户个人信息，显示欢迎用户页面。

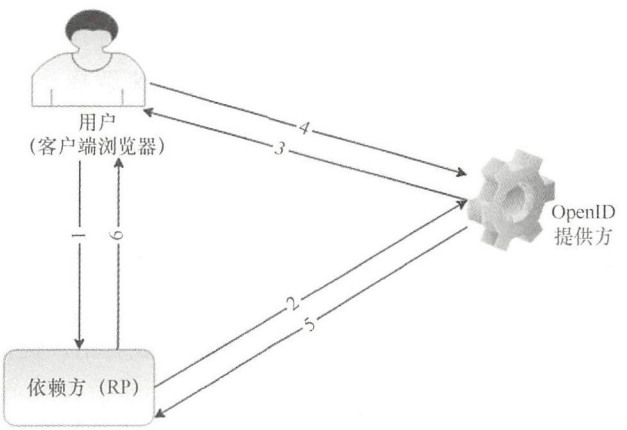
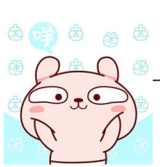


图 4-1

这是 OpenID 提供方的一个简单的工作流。所以不管用户访问多少个网站，只要每次都选择同一个 OpenID 提供方，那么用户就只需要记住一个用户名/密码就可以了。这就是 OpenID 带来的便捷。

4.2.2 OAuth 2.0

OpenID 可以进行用户权限验证。在微服务中，除了登录之外还面临权限验证的问题。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务中每个服务分管一个领域的业务逻辑，那么是不是每个成功登录的用户都应该能访问所有的微服务呢？如果是这样的话，可以把用户和所有微服务混到一起，也就是与单体应用的方式差不多。这种方式是有问题的，因为一旦攻击者突破了登录这道安全关卡，整个微服务系统就暴露无遗了。

这就意味着每个用户/客户端的权限应该是有限的。不是每个用户/客户端都能访问所有的微服务或者使用某个微服务中的所有操作。在这种架构下，每个服务在执行操作之前都应该用这样一种方式，来检验用户权限。每个客户端/用户向调用微服务时需要出示权限凭证，该凭证能证明它有执行该操作的许可权。OAuth 就是用来解决这个问题的一种实现方式。

OAuth 2 是一个授权框架。应用可以通过 OAuth 2 获得 HTTP 服务上某些资源的有限访问权。OAuth 的概念的提出始于 2006 年。当时 Twitter 和 Gnolia 的开发人员正在讨论如何认证因特网服务，在当时并没有现成的 API 访问代理的标准。他们随后于 2007 年 4 月在网上成立了一个 Oauth 委员会，并且起草了 OAuth 方案的草案。随后这个 OAuth 的这一草案被纳入了互联网工程任务组（IETF）中。

OAuth 2 与认证和登录过程不同。虽然世界范围内有很多不同的人努力让 OAuth 2 可以高效地独立使用，但是现阶段普遍被人接受的方案是使用 OpenID 来做认证，并使用 OAuth 2 来做授权。我们来举个实际生活中的例子来解释一下二者：一位访客来到学校会见一名学生，该访客可能是该学生的某个亲戚，那么访客与学生会面的流程如下。

（1）访客来到学校的接待处，说明他想见到某个学生。

（2）学校接待处会给该学生发一个请求，包含访客提供的身份信息。

（3）学生来到接待处，确认访客是自己的亲戚。

（4）直到此时都是 OpenID 做的事，当任何一个服务或者用户提供了认证信息之后，OpenID 会验证这些认证和身份信息是否正确。

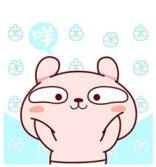
（5）访客会拿到一个访客的门卡。

（6）现在该学生和该访客可以在一个会议室进行会面，如在父母房里。

现在访客可以通过学校的接待室，但是他只能待在学校里有限的几个地方，如父母房或者自助餐厅等，但是别的地方是不准去的。这就是 OAuth 做的事，OAuth 是一个用于授权的代理协议。OAuth 也可以用来认证用户，Facebook 就使用它来做认证，但是 OAuth 的主要用途是做授权的。

OAuth 中主要有 4 种角色：

- 资源所有者；
- 资源服务器；



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- 授权服务器；
- 客户端。

资源所有者是一个特殊用户，它可以给其他账号分配权限。资源服务器持有 RO 需要使用的资源。授权服务器是用来授权和验证身份的。客户端可能是想要访问某个账号的第三方的应用。

最开始的想法是用户可以通过 OAuth 提供方服务器注册应用，注册之后服务器会提供客户端的 ID 和密码。这个流程如下所示。

（1）用户尝试访问客户端应用的资源，这可能是一个网页。

（2）客户端将用户（RO）重定向到认证服务器，客户端同时还会把其自身的客户端 ID 和密码一并加到请求中。

（3）用户（RO）授权客户端应用获取用户个人信息。

（4）认证服务器将用户的授权许可返回给客户端。

（5）客户端会为用户从认证服务器获取一个令牌。

（6）资源所有者根据这个令牌来将资源的访问权限授予该用户。

4.2.3 JWT

JWT 是不同服务之间通信时的一种特殊的令牌格式。行业内有了很多专利标准，但是被广为接受的令牌标准，JWT 恰好填补了这一空白。JWT 定义了一种标准化的令牌格式。这种格式的详细信息可以在 RFC 文档中找到，地址是 https://tools/ietf/org/html/rfc7519。JWT 除了包含认证信息外，还包装了一些元数据。通常用户的名字和权限范围会放入 JWT 中。你也可以往 JWT 中添加一些自定义的元数据。

OpenID 和 OAuth 是用来认证和授权的框架。这二者讨论的都是访问信息的概念，但是它们都没有谈到令牌的加密和格式的问题。JWT 只是一种令牌应该如何包装和加密的规范。每个组织可以有自己的服务器和一些不同的令牌格式。JWT 也是一种广泛被接受的标准化令牌格式。

JWT 有其自身的结构，分为几个部分，大体来讲有以下 3 个部分：

- 头部部分；
- 净荷部分；
- 签名部分。

这些部分在 JWT 令牌中以“.”来分隔。例如，一个 JWT 令牌可能是以下的形式：

- `EyJ0eXAiOiJKV1QiLCJhbGciOiJS`（第一部分，头部）；
- `e-KqaMUWvxjR1VJyI8hzbrGV65fTvHY`（第二部分，净荷）；

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- U3gNnYT1KcrA22OYsJyvhKhTEE_Ztx（第三部分，签名）。

这 3 部分是通过点连接在一起的，没有换行。这样解析的时候就能根据“.”将令牌解析为这 3 部分了，每个部分应该是单独加密的。

几个例子

如果我们解码 JWT 令牌，其具体结构应该是下面这样的。

（1）**头部部分**。在这一部分会用到下面这些项。

- alg：这是一个字符串类型的参数，用于定义 JWT 加密数据时使用的算法，如 RS256。
- typ：这是一个字符串类型的参数，用于描述令牌的结构。现在该参数的值应该是 JWT。这个字段现在不是必需的，将来如果使用一些其他结构的时候或许可以派上用场。
- kid：这是一个字符串类型的参数，用于提示验证签名时应该使用什么样的密钥。
- x5u：这是一个字符串类型的参数，用于指定使用 URL X.509 认证公钥来验证签名
- x5t：这也是一个字符串类型的参数，为 X.509 认证提供 SHA-256 字节的 DER 编码。它是使用 base64 编码的。kid 参数可以使用下面这样一个值：

```
{
   "typ": "JWT",
   "alg": "RS256",
   "x5t": "ywSXXXXEiJkEXXXXXXXXXXK9SK-k",
   "kid": "ywSXXXXEiJkExXXXXXXXXXK9SK-k"
}
```

（2）**净荷**。净荷部分的字段既有标准字段，也支持自定义字段。其中包含的一些标准字段解释如下。

- iss：指的是发布 JWT 令牌的标识。
- aud：指的是令牌的接收方的值。
- sub：指的是令牌的主题，这个字段不是必需的。
- exp：指的是过期时间，超过该 Unix 时间之后，令牌不再有效。
- nbf：指的是不早于时间，指的是该 JWT 可以用于接受和处理的时间不得早于该时间。
- iat：指的是签发时间，通常是 JWT 令牌签发的 Unix 时间。

以上这些就是 JWT 令牌中使用的一些标准参数。下面我们以一个例子来讲解如何利

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

用 JWT 令牌来使应用更加安全：

```
{
  "iss": "http://YourIdentitySever.com",
  "aud": "http://YourIdentitySever.com/resources",
  "exp": 1481576528,
  "nbf": 1481572928,
  "client_id": "84802e3f-7fde-4eea-914e-b9e70bdfc26f",
  // you can have any custom data here ,
 "scope": "admin_services"
}
```

4.2.4 示例应用

现在我们已经对 JWT 令牌有了一个基本的了解，我们还可以自己实现或者找一些已有的库来解析 JWT 令牌中的信息。本节的示例会用到 Spring Boot 和 Spring Security 的功能。本下面，我们将创建一个自定义的过滤器，并将其放在 `UsernamePassword AuthenticationFilter` 之后。这个过滤器会从请求头部中提取令牌信息，然后做很多认证方面的检查。首先会检查 JWT 令牌是否有效，其次会检查令牌的权限和发行方。这些检查如果失败会产生相应的消息。假设任何一个认证服务都可以生成 JWT 令牌。认证服务也可以检查自己生成的令牌。在前面展示的示例中，我们并没有用 Java 实现任何认证服务。在这里会讲解这部分客户端的代码实现。

示例会使用来自 bitbucket 的 jose4j 的库。使用这个库需要在 POM.xml 文件中添加下面这样一个依赖：

```
<!-- For JWT Filter -->
    <dependency>
        <groupId>org.bitbucket.b_c</groupId>
        <artifactId>jose4j</artifactId>
        <version>0.5.1</version>
    </dependency>
```

同时还要在应用中用到 Spring Security，因此也需要在 POM.xml 文件中添加 Spring Security 的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

然后创建两个不同的 URL 请求，一个是安全的，另一个是不安全的：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@RestController
@RequestMapping("/PM/")
public class SecurityDemoSecureController {

    @RequestMapping(method = RequestMethod.GET, value = "/secure/greet",
    produces = "application/json")
    public String getSecureHello() {
        return "hey! secure";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/unsecure/greet",
    produces = "application/json")
    public String getUnSecureHello() {
        return "hey! unsecure";
    }
}
```

以上就是控制器类中的代码。接下来给示例应用创建一个配置文件：

```
@SpringBootApplication
public class SecurityDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecurityDemoApplication.class, args);
    }
}
```

在传统的 Spring Security 实现中，通常要有一个 AbstractSecurityWebApplicationInitializers 类。这里因为使用了 Spring Boot 的内嵌容器，所以不再需要 WebApplicationInitializers 接口。

Spring Security 会有另外一个配置类，虽然可以把 Spring Security 的配置和前面体到的主应用的配置合并到一起，但是简单起见，还是把它们两个分开实现：

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    privateJwtEntryPointunauthorizedHandler;

    @Autowired
    privateJwtVerificationServicejwtVerificationService;

    @Autowired
    public void configureAuthentication
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
(AuthenticationManagerBuilderauthenticationManagerBuilder) throws
Exception {
    authenticationManagerBuilder.userDetailsService
    (this.jwtVerificationService);
}

@Bean
@Override
publicAuthenticationManagerauthenticationManagerBean() throws
Exception {
    returnsuper.authenticationManagerBean();
}

@Bean
publicAccessDeniedHandlergetJwtAccessDeniedHandler() {
    JwtAccessDeniedHandler handler = new JwtAccessDeniedHandler();
    return handler;
}

@Bean
publicJwtAuthenticationTokenFilterauthenticationTokenFilterBean()
throws Exception {
    JwtAuthenticationTokenFilterauthenticationTokenFilter = new
    JwtAuthenticationTokenFilter();
    authenticationTokenFilter.setAuthenticationManager
    (authenticationManagerBean());
    return authenticationTokenFilter;
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.sessionManagement().sessionCreationPolicy
    (SessionCreationPolicy.STATELESS).and().authorizeRequests()
    .antMatchers(HttpMethod.GET,"/PM/secure/**").
    access("hasAnyRole('ROLE_Secure_Access')")
    .antMatchers(HttpMethod.POST, "/PM/secure/**").
    access("hasAnyRole('ROLE_Secure_Acces')").and()
    .exceptionHandling().accessDeniedHandler(getJwtAccessDeniedHandler())
    .authenticationEntryPoint(unauthorizedHandler);
```

接下来需要在内置过滤器 UsernamePasswordAuthenticationFilter 之前添加自定义过滤器：

```
    http.addFilterBefore(authenticationTokenFilterBean(),
    UsernamePasswordAuthenticationFilter.class);
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        http.csrf().disable();
        http.headers().cacheControl();
    }

    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/PM/unsecure/**");
    }
}
```

如上所示，UsernamePasswordAuthenticationFilter 之前添加了一个新的过滤器，名为 JwtAuthenticationTokenFilter。Spring Security 会在调用 UsernamePassword 过滤器之前调用这个过滤器。这个过滤器会扫描请求头部，然后尝试从认证头部信息中提取令牌信息。这里有一个假设是令牌会跟着令牌类型一起放在请求中发给后台。这也是我们将头部的值按照空格进行了拆分的原因。一般来讲，头部中的令牌应该是类似于以下这样：

```
"bearer
yJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Inl3U1ZJZ0VpSmtFeHJIRTN5b29jS0s5
U0stayIsImtpZCI6Inl.....................JhKhTEE_ZTxE-
QwkIfbHIm8LBm60DwpEYCZvyPH-kAtr7bCl-A  "
```

这里 bearer 就是表示令牌类型的。令牌类型和令牌值之间是通过空格进行分隔的。

过滤器类会提取令牌信息，然后将令牌信息发送给令牌验证服务进行验证。下面是自定义过滤器类中 doFilter 方法的实现代码：

```
@Autowired
AuthenticationManagerBuilder authBuilder;

//Spring auto wired Default Authentication Builder instance to this class.

@Override
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
throws IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest) request;

    //Get the authorization header form request.

    String header_authorization =
    httpServletRequest.getHeader("Authorization");
    String token = (StringUtils.isBlank(header_authorization) ? null :
    header_authorization.split(" ")[1]);
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        if (StringUtils.isBlank(header_authorization) && token == null) {
            logger.info("Token Not found in header .");
            return;
        }
        UserDetails principal = null;
        try {
            principal =
            authBuilder.getDefaultUserDetailsService().loadUserByUsername(token);
            UsernamePasswordAuthenticationToken userAuthenticationToken = new
            UsernamePasswordAuthenticationToken(principal, "",
            principal.getAuthorities());
            userAuthenticationToken.setDetails(new
            WebAuthenticationDetailsSource().buildDetails(httpServletRequest));
            SecurityContextHolder.getContext().
            setAuthentication(userAuthenticationToken);
        } catch (Exception e) {
            HttpServletResponse httpresposne = (HttpServletResponse)response;
            httpresposne.setContentType("application/json");
            httpresposne.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
```

这里可以创建一个 JSON 格式的响应来展示一个返回给用户的响应结构：

```
            ObjectMapper jsonMapper = new ObjectMapper();
            PrintWriter out = httpresposne.getWriter();
            Map<String, String> jsonResponse = new HashMap<String, String>();
            jsonResponse.put("msg", "Invalid Token");
            out.write(jsonMapper.writeValueAsString(jsonResponse));
            out.flush();
            out.close();
            return;
        }
    }
    chain.doFilter(request, response);
}
```

验证服务需要 3 个属性，即 jwksBaseURL、jwksIssuer 和 jwksAudience。认证服务需要这 3 个属性来验证令牌。在验证令牌之后，这个服务会检查令牌中提到的用户，然后设置其身份，然后将其发给后面的过滤器。紧接着过滤器会检查令牌中的角色是否有足够的权限来访问其请求的对象。如果没有足够的权限，会返回无效令牌作为响应：

```
@Service
public class JwtVerificationService implements UserDetailsService {
    private static final Logger logger =
    Logger.getLogger(JwtVerificationService.class);
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
private static final String CLASS_NAME = "JWTVerificationService";

public static final String SCOPE_PREFIX = "ROLE_";
```

下面这几个就是可以从属性文件中注入的配置项：

```
@Value("${JwksUrl}")
protected String jwksBaseURL;

@Value("${JwksIssuer}")
protected String jwksIssuer;

@Value("${JwksAudience}")
protected String jwksAudience;

@SuppressWarnings("unchecked")
@Override
publicUserDetailsloadUserByUsername(String token) throws
UsernameNotFoundException {
    String username = "";
    String role = "";
    JwtClaimsjwtClaims = null;
    List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
    try {
        jwtClaims = getJwtClaims(token);
        username = (String)
        jwtClaims.getClaimsMap().get("client_id");
        logger.debug(CLASS_NAME + "userName :" +
        jwtClaims.getClaimsMap().get("client_id"));
        role = (String) jwtClaims.getClaimsMap().get("scope");
        authorities.add(new SimpleGrantedAuthority(SCOPE_PREFIX + role));
        logger.debug(CLASS_NAME + "JWT validation succeeded! With
        Scope " + role);
    } catch (ClassCastException e) {
        logger.debug("Not able to type cast Scope in String , Trying
        with array list for multiple Scope.");
        if (jwtClaims != null) {
            List<String>roleList = (ArrayList<String>)
            jwtClaims.getClaimsMap().get("scope");
            for (String roleStr : roleList) {
                authorities.add(new
                SimpleGrantedAuthority(SCOPE_PREFIX + roleStr));
            }
            logger.debug(CLASS_NAME + "JWT validation succeeded!
            with Scope " + authorities);
        }
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        } catch (Exception e) {
            logger.debug("Invalid JWT !!! token = {"+token+"} found
            and exception = ", e);
        }
        return (username != null &&username.length() > 0) ?
        (UserDetails) new User(username, "", authorities) : null;
    }

    private JwtClaimsgetJwtClaims(String token) {
        HttpsJwkshttpsJkws = new HttpsJwks(jwksBaseURL);
        HttpsJwksVerificationKeyResolverhttpsJwksKeyResolver = new
        HttpsJwksVerificationKeyResolver(httpsJkws);
        JwtConsumerjwtConsumer = new
        JwtConsumerBuilder().setRequireExpirationTime().
        setAllowedClockSkewInSeconds(3600)
                .setExpectedIssuer(jwksIssuer)
                // whom the JWT needs to have been issued by
                .setExpectedAudience(jwksAudience).
                setVerificationKeyResolver(httpsJwksKeyResolver).
                build();
        try {
            // Validate the JWT and process it to the Claims
            JwtClaimsjwtClaims = jwtConsumer.processToClaims(token);

            return jwtClaims;
        } catch (InvalidJwtException e) {
            // Anyway here throws the exception , so no need to log the
            // error.
            // log the error if required from where this function
            // invokes
            // logger.error("Invalid JWT! " + e);
            throw new AuthenticationServiceException("Invalid Token");
        }
    }
} Credit-scoring microservice
```

示例的信用评分服务可以基于以下几个因素来计算评分：

- 支付历史占 35%；
- 当前贷款占 30%；
- 付款历史时长占 15%；
- 个人债务占 10%；
- 近期信用活动占 10%。

因为假设支付微服务已经实现，所以支付历史的信息应该由支付微服务来提供。简

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

单起见，如果最高分是 100 分的话，个人债务服务能给出的分数就是 0～10 分，这样就不需要归一化之后再进行加权处理。债务本身有一些计算规则，如表 4-1 所示，假设当前个人是家庭中唯一的收入获得者。

表 4-1

薪水 家庭负担	<1 万	1 万～2 万	2 万～4 万	4 万～6 万	6 万～8 万	8 万～10 万	≥10 万
独居	2	3	4.5	5	6.6	7.4	8.9
朋友	1.8	2.8	4.3	4.8	6.4	7.2	8.8
配偶	1.4	2.6	4.1	4.6	6.1	6.9	8.6
配偶和孩子	1.2	2.1	3.8	4.2	5.6	6.6	7.9
配偶孩子和父母	1	1.9	2.8	3.9	5.1	6.4	7.5

记住，表 4-1 里都是一些假数据，具体的数值会随着数据模型以及所处的国家有很大不同。对于示例应用而言，不需要在意这些数据的准确性。

这应该是另外一个 Spring Boot 应用。这里假设所有其他微服务都在使用 JWT 作为安全验证。因此，可以使用某一个服务来获取 JWT 令牌，然后将令牌包装在请求中，然后以此令牌去别的服务获取数据。

示例的 POM 文件中使用了两个新的依赖：一个用来将 JSON 响应转换成一个 Java 类对象，另外一个是 Apache 的用 Java 代码来发 REST 请求的 httpclient 库：

```
<dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.5.2</version>
    </dependency>
    <!-- For json to Object Conversion -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.2.4</version>
    </dependency>
```

控制器类只有一个方法，根据具体的用户 ID 来获取其信用评分。这个类的代码非常少：

```
package com.practicalMicroservcies.contoller;
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@RestController
@RequestMapping("/PM/credit/")
public class CreditScoringController {
    private static final Logger logger =
    Logger.getLogger(CreditScoringController.class);

    @Resource
    CreditScoringServicescreditService;

    @Resource
    ObjectMapper mapper;

    /**
     * Method is responsible for getting the account detail for given ID.
     *
     * @paramuserId
     * @return
     */
    public static final String getScore = "getScore(): ";
    @RequestMapping(method = RequestMethod.GET, value = "score/{userId}",
    produces = "application/json", consumes = "application/json")
    public ResponseEntity<String> getScore(@PathVariable("userId") UUID
    userId) {
        logger.debug(getScore + " getting information for userId " + userId);
        return new ResponseEntity<>(creditService.getUserScroe(userId),
        HttpStatus.OK);
    }
}
```

此外，还应该有一个服务类来获取令牌，与其他不同的服务对话，从不同的服务处获取数据，最终计算出总的信用评分，之后打包成响应结果返回。这个特殊的服务类需要下面这些属性：

```
@Value("${AuthServerUrl}")
    protected String authServerUrl;
    @Value("${ClientId}")
    protected String clientId;

    @Value("${ClientSecret}")
    protected String clientSecret;

    @Value("${GrantType}")
    protected String grantType;
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
@Value("${paymentHistoryUrl}")
protected String paymentHistoryUrl;

@Value("${obligationUrl}")
protected String obligationUrl;

@Value("${loanUrl}")
protected String loanUrl;
```

这个类中的这些属性的定义如下所示。

- `authServerUrl`：这是认证服务器的完整 URL。
- `clientId`：这是认证服务器分配给当前服务的 ID。
- `clientSecret`：这是认证服务器提供的客户端密码，是在从认证服务器获取令牌时和 `clientId` 一起产生的。
- `grantType`：这是访问其他服务时的访问级别控制，如读写权限等。你也可以定义自己的允许类型。
- `paymentHistoryUrl`、`obligationUrl` 和 `loanUrl`：这几个属性顾名思义，表示的是这些服务的 URL。

现在，让我们来看一下这个从认证服务器获取令牌的方法是怎么实现的。这个方法将所有值都放在请求的请求体部分，然后以 POST 的方式将请求发送给了认证服务器。在响应中，假设拿到的 JWT 令牌是一个字符串，通过这个令牌可以从别的服务处获取数据：

```
private String getToken() {
    RestTemplaterestTemplate = getRestTemplate();
    MultiValueMap<String, String> map = new LinkedMultiValueMap<String,
    String>();

    map.add("client_id", clientId);
    map.add("client_secret", clientSecret);
    map.add("grant_type", grantType);

    String tokenStr = restTemplate.postForObject(authServerUrl, map,
    String.class);
    return tokenStr;
}
```

这里有一个小的方法叫 `getRestTemplate()`，这个方法用到了一个名为 `bufferingClientHttpRequestFactory` 的 HTTP 工厂类。这是因为，如果想用日志记录下请求和响应中的关键信息，就需要通过缓存 HTTP 请求客户端来初始化

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

restTemplate 对象：

```
private RestTemplategetRestTemplate() {
    ClientHttpRequestFactory simplerequestFactory = new
    HttpComponentsClientHttpRequestFactory(
    HttpClients.createDefault());
    ClientHttpRequestFactoryrequestFactory = new
    BufferingClientHttpRequestFactory(simplerequestFactory);
    RestTemplaterestTemplate = new RestTemplate(requestFactory);

    return restTemplate;
}
```

下面这个方法中会访问 3 个不同的 URL，然后返回接收到的响应：

```
/**
 * Method is responsible for getting the Payment History
 *
 * @paramuserId
 * @return
 */
private List<String> getPaymentHistory(UUID userId) {
    String token = getToken();
    if (token == null)
        return new ArrayList<String>();
    RestTemplaterestTemplate = getRestTemplate();

    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "bearer " + token);
    headers.add("Content-type",
    ContentType.APPLICATION_JSON.getMimeType());
    headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
    HttpEntity<String> request = new HttpEntity<>(headers);

    ResponseEntity<String> result =
    restTemplate.exchange(paymentHistoryUrl + userId, HttpMethod.GET, request,
    String.class);
    ArrayList<String>responseObject = new
    Gson().fromJson(result.getBody(), ArrayList.class);
    return responseObject;
}
/**
 * Method is responsible for getting the Obligation scrore
 *
 * @paramuserId
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
     * @return
     */
    private Integer getObligation(UUID userId) {
        String token = getToken();
        if (token == null)
            return 0;
        RestTemplaterestTemplate = getRestTemplate();

        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", "bearer " + token);
        headers.add("Content-type", ContentType.APPLICATION_JSON.getMimeType());
        headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
        HttpEntity<String> request = new HttpEntity<>(headers);

        ResponseEntity<String> result = restTemplate.exchange(obligationUrl +
        userId, HttpMethod.GET, request, String.class);
        return result.getBody() == null || result.getBody().length() ==
        0 ? 0 :Integer.parseInt(result.getBody());
    }

    /**
     * Method is responsible for getting the Obligation loan score
     *
     * @paramuserId
     * @return
     */


    private Integer getLoansList(UUID userId) {
        String token = getToken();
        if (token == null)
            return 0;
        RestTemplaterestTemplate = getRestTemplate();

        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", "bearer " + token);
        headers.add("Content-type", ContentType.APPLICATION_JSON.getMimeType());
        headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
        HttpEntity<String> request = new HttpEntity<>(headers);

        ResponseEntity<String> result = restTemplate.exchange(obligationUrl +
        userId, HttpMethod.GET, request, String.class);
        return result.getBody() == null || result.getBody().length() ==
        0 ? 0 :Integer.parseInt(result.getBody());
    }
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这里的所有方法都是用来从不同的服务获取数据的。所有获取数据的请求都是安全的，因为我们采用了 JWT 和相应的加密方式。有了所有这些数据之后，就可以生成一个简单的信用评分值了。债务分数可以拿过来直接用。支付历史可以通过识别失败和逾期支付的数量来估算，如历史中有 56 笔失败的支付等，可能会需要有一个类似债务分数计算表的映射表来计算支付历史对应的评分。在这个例子中，这 56 笔失败的支付历史来自总共 500 次支付中，也就是说，超过 10%的支付都是失败的。我们可以基于这个 10%的比例和一些别的因素来给出一个较为合理的评分，比如说这 10%支付失败的黑历史要扣除掉总分中的 45%，那么就是说支付历史这一项最后总得分是 100-45=55 分（占整个支付历史总分的 55%），然后整个支付历史这一项的加权分是 35%，因此相乘之后 55%×35%约等于 19.25%。在结合其他几个因素的评分之后，就能很容易地给出用户的最终总评分了。

前面提到的这个计算公式只是一个假的例子。在理想情况下，这个计算公式需要在更多的统计数据和一些机器学习算法以及大量数据分析的支撑下才能比较准确。这里只是用这个简单的信用评分应用举个例子。如果数据建模更加高效，信用评分过程也会更加高效。

4.3 小结

本章讨论了微服务所面临的安全挑战，以及在微服务架构中应该遵从一些什么样的安全实践。本章还讨论了 OpenID 和 OAuth 2.0。此外，本章还引入了 JWT 令牌的认证方式，并且将其用到了示例项目中。下一章则会着重讨论一种高效的数据模型以及示例应用中所使用的数据模型。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 5 章
创建高效的数据模型

软件行业日新月异，软件开发中的规则、维度和准则每天都在发生着变化。想要了解行业内的更多新知识，需要反复迭代构建、收集反馈然后再提升。单体架构的环境下这种迭代会越来越复杂。而如果选用了微服务架构，整个过程就会流畅很多。微服务的思路是把复杂问题拆解为一些小的问题然后逐个解决，按照这个思路我们可以把功能拆解为一些分布式组件。在一个分布式以及有一定扩展性要求的领域工作时，总是要挑战传统做事方式，这种效应可以在数据库方面看出来。传统的数据模型也无法用于分布式架构中。另外一个挑战是数据之间的依赖关系，随着问题逐渐细化和变成分布式的，数据模型、代码库等之间的依赖关系应该如何管理是很有挑战的。什么样的数据库模型才是最适合微服务的？针对这个问题没有一个放之四海皆准的标准答案，但是接下来的这一章中将会着重讨论选择数据模型时应该考虑的一些重要因素。

本章中将会覆盖下列主题：

- 微服务数据模型结构；
- 与传统数据模型的对比；
- 必要时的混合数据技术；
- 一个应用数据模型的示例。

5.1 数据和建模

现如今的数据可以说遍地都是。每一个业务都需要基于数据运行，与此同时分析历史数据也能更好地为业务当前及长期的发展策略提供最有力的支撑。数据是很重要的，但是对于某一个具体的业务而言，并不是所有的数据/信息都是相关的和有用的，所以对于软件系统而言，它既不会用到所有的数据，也同样不会存储所有的数据。开发人员需要识别出哪些数据对于业务是很重要的，并只存储这部分数据，然后将这部分重要的数据作为未来的参考。那么，识别到重要的数据之后，应该以何种方式对其进行存储呢？

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

是应该用数据原来的格式存储还是应该定义一个结构来进行存储呢？要想搞清楚数据的意思和分析数据提取数据所反映的重要信息，一般需要把数据按照一定的结构进行存储。识别和定义数据结构的过程就是数据建模。

维基百科对于数据模型的定义是：一种组织数据元素的抽象模型，以一种标准化的方式定义了数据元素之间的相互关系以及各自对应到物理世界实体的属性。

数据建模是组织数据语义的可视化表示。它包含业务涉及的实体，如客户、产品、订单等。数据建模还包含建立这些实体之间的连接关系。实体及其属性是数据库中存储的最主要的因素。举个例子，对于客户而言，通常会有这样一些属性：客户名字、地址、电话号码等。另外一个重要因素是基数，也就是关系的维度。例如一个客户可以订购多个产品，也可以下多个订单，所以此种情况下客户和订单之间是一对多的关系。另外，还有一些多对一、多对多的基数关系等。

换句话说，可以这么理解数据模型：数据模型是对复杂真实世界数据的一种相对简化的表达。对于复杂现实世界的理解仁者见仁，智者见智，因此一个好的数据模型应该能满足多种不同权限用户的需求。

与传统数据模型的对比

在一段时间内，软件工程产生了一些演化。在演化的过程中也带动数据库建模和数据库设计产生了一些演化。从单体架构到基于服务的架构的转变过程中，数据库建模也完成了不同的里程碑。在这一节里，我们会探讨单体应用和 SOA 架构中的数据库建模是如何做的，然后会讨论微服务架构中数据库建模应该怎么做。需要记住一点，我们这里讨论的建模技术并不是唯一的，也不会强烈推荐使用哪种建模技术。这里所讨论的建模技术只有为了给读者一些概念，即大多数情况下，大家是怎么做数据建模的。这些普适的建模方式也被广泛地用在很多系统中。

5.2　单体架构中的数据模型

在单体应用中，所有模块都揉在一个大的应用中，通常是一个 WAR/JAR/EAR 包。在单体应用中，如果一个模块需要使用另外一个模块的功能，它只需要简单地调用那个模块就行了。如此一来就建立了一些模块之间的依赖。模块与模块之间共享一些数据。如果微服务也按照这种方式，各个服务使用同一个数据库实例，那么最终也会通过一个公用数据模型使各个模块紧耦合在一起，形成依赖。这也是我们说数据模型可以看作应用背后一个庞大的数据库的原因。在传统的数据模型中，我们通常会从一个实体关系图出发，最终设计出一个关系模型。这个关系模型包括表结构和表与表之间的关系。属性

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

对应到这些表中的字段或者列名。关系型数据库表中的一行称为一个元组，每个元组必须有一个唯一的键来确保数据的唯一性。这种键被用来建立表与表之间的关联关系。关系型模型中还有外键的概念，通过外键可以很方便地建立关系和连接。

图 5-1 给出的是一个简单的例子，假设的场景是某个给学生提供课程的组织想要记录信息。第一步是我们要识别出系统中的重要实体，如课程、学生、科目和老师等。一名学生可以登记一门课程，一个课程可以有一个或多个科目。一位老师可以教一个科目（目前是这样）。这样就可以识别出来这些实体之间的关系。课程表会存储与课程相关的数据，学生关系会存储某个学生登记某门课程的数据。每个课程和科目的映射关系，通过一个 `map-course-subject` 表来存储。这是一个非常简单的关系模型的例子。

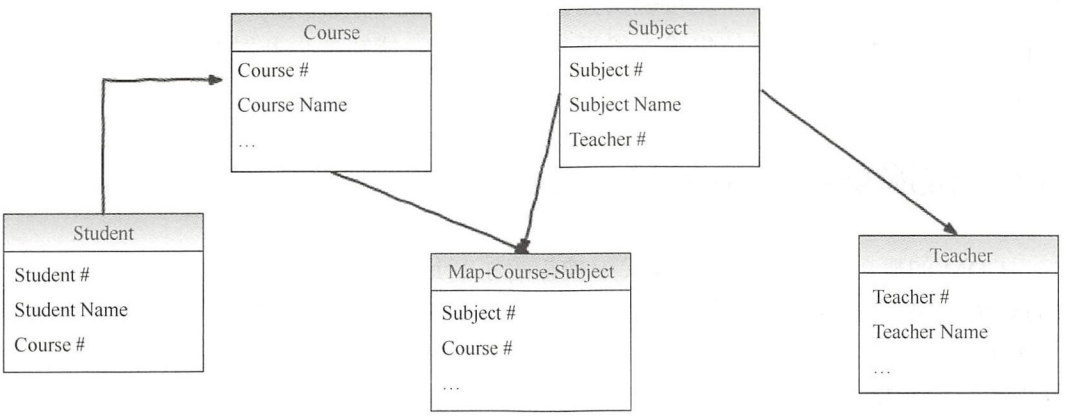


图 5-1

数据是由一些实体以及实体间关系组成的。关系有两种，即 IS-A 和 HAS-A。例如，课程与科目就是一种 HAS-A 的关系。HAS-A 关系表示一个实体（课程）内部必须有另外一个实体（科目）才算完整。而 IS-A 关系所描述的两个实体通常是同类，例如，沃尔沃是一种汽车，所以沃尔沃和汽车之间的关系是 IS-A。

前面的场景中提到的这个方式非常好。但是因为实体之间的关系非常多，数据需要都放在一个地方，这样才能获得有意义的数据。这样就可以执行关系间的联合查询，外键等约束也可以在不同的表中使用。使用这种方式时，应用的所有数据是紧耦合的。修改数据模型时需要重新发布整个系统。

5.3 SOA 中的数据模型

前面提到的这种数据模型通常用在一些传统的单体应用架构中。下一步来看看 SOA，也是一种有代表性的面向服务型架构。这里会提到一种新的数据模型，名字是规

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

范化数据模型（CDM）。SOA 使用的就是这样一种数据模型。这个模型鼓励在不同系统和组件之间传递消息时，尽可能重用数据结构、属性和数据类型。这种方式的好处是可以简化业务数据的反复定义，但是这个方式很容易变得不可管理。不管你选用的是哪一种技术（如 ESB 或者 BPM 平台或者某种服务的组装），都需要标准化所传输的业务对象的数据模型。在 OOP 的理念中，通常会有多种大家一致定义的实体，如客户、订单、产品等，还会有一堆 ID、属性和联系。有了这种标准化的数据模型之后，看起来似乎大家用起来都很方便了，但是如果某个人的数据模型与这个标准化的模型不同，他就需要自己实现一些转换器，来确保标准化模型和自己的模型可以互转。

使用标准化数据模型有一些好处，例如 CDM 会促进组织内的各个业务部门的人使用一种通用的对象定义。这可确实能改善沟通和相互理解，也能减少产生歧义和误解的机会。与此同时，系统集成变得更加简单高效，能够减少开发时间和开发人员的工作量。

5.4　微服务架构中的数据模型

CDM 的概念在微服务集成模式中被拒绝了，原因是微服务根本就不需要 CDM。CDM 鼓励用户将数据共享到一个公共渠道。如果我们沿用了 CDM 的方式，用户会不由自主地在所有微服务中使用同样的技术，这与我们开始提到的微服务的多语言性质是相违背的。而且如果你的数据模型是共享的，那么用到这些数据模型的微服务是耦合在一起的，这样一来会丧失微服务的一个重大好处：各个开发团队不再能够不受限并无须知道其他微服务团队的开发而独立开发。关于微服务的数据模型，我们重点要记住的是以下两条原则。

- **松耦合**：任何微服务的修改都不会影响其他任何服务。
- **问题定位**：发现的问题应该归类到一起。

基于这两个原则，为了做到松耦合，我们需要一个数据模型的明确的契约。

微服务的数据建模方式是分布式和相互独立的。作为微服务的一个原则，每个服务都是独立的、隔离的。这个原则同样会影响到数据库建模。如果问题分析和微服务拆分做得非常好的话，每个服务是可以并行地独立开发和部署的。在这种理想状态下，每个微服务的数据模型也应该与其他服务的数据模型相互独立。因此，任何一个微服务数据模型的修改，均不会对其他服务中的数据模型产生任何影响。

5.4.1　每个微服务限定一些数据库表

在这种情形下，我们很容易想到应该给每个微服务分配一些私有数据。只有对应的

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务可以访问、更新和修改其私有数据。具体的实现方式有很多种。我们可以给每个服务分配一组数据库表，这一组表中的数据只能被这个特殊的微服务修改。要实现这种模式，我们可以给服务分配一个特殊用户，只有这个用户有权限修改这些表。采用这种方式我们就能确保每个微服务不会干扰与其他微服务相关的表中的数据。

把所有的表都放在一个数据库中，然后通过数据库用户权限来控制各自服务的表的访问权限听起来是个不错的想法，但是这种方式也还是有问题，它会将我们所有的微服务限定到同一种类型的数据库中，如 Oracle 或者 MySQL 数据库。如果某个团队想尝试一种不同类型的数据库来改善性能，在这种情形下是不可能的。

5.4.2 每个微服务一个数据库

另外一种方式是每个微服务拥有一个单独的数据库。这种方式可以解放开发人员的双手，使他们可以根据各自微服务的需要选择适合的数据库类型。开发人员无须局限于其他微服务使用的一种数据库技术。这种方式看起来像是一种分布式的 CDM，似乎很有说服力，但是也还是面临着一些挑战。各个服务的共享数据以及流经不同服务的数据还是紧耦合的。要使系统保持一致状态，必须执行所有的命令（可能涉及多个微服务）来填充请求，而如果任何一个中间步骤失败，必须要能够正确回滚到原始状态。在传统数据模型中，这是相对简单的，因为传统数据模型中有一个表上的锁，提交和回滚都是发生在一个数据库中。在分布式环境中，每个组件都是单独运行的，并且每个组件都可以从任一中间状态回滚到最初状态。

举个例子，商业交易过程（如财务交易）通常会涉及多个微服务，某些交易发生的时候多个微服务都需要更新数据。此种情况下高效地处理故障切换非常有挑战。除了使用分布式事务之外，一种解决办法是你通常必须使用最终一致性的事件驱动的方式来保持数据库的一致性。这里我们推荐的一种方式是使用 Saga 模式。这种模式最开始是 Hector Garcia-Molina 和 Kenneth Salem 在 1987 年开始使用的。他们设计这种模式时一直牢记要保持一个非常松耦合的环境。Saga 看起来是一个非常吸引人的模式。Saga 模式虽然没有保证事务回滚成初始状态，但是它可以保证系统一致性，可以回滚到一种非常接近初始状态的状态。除了 Saga 之外，可以考虑分布式环境中两阶段提交的方式，但是这种方式在可扩展环境中似乎不是很理想的解决方案。

5.4.3 Saga 模式

Saga 模式提供了一种在分布式环境中管理事务的解决方案。分布式环境中会有一些相互通信的不同子系统（不同的微服务）。在真实世界中，分布式环境中通常会有一些历

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

时长久的事务。数据库中的事务指的是要么全部成功要么全部失败，全部成功的情况较好处理，系统会自动变成一致的。难就难在某个子组件的事务失败后，整个事务如何整体回滚到一致的状态。

在 Saga 中，每个事务的工作流包含一组混合任务，被称为传送名单。传送名单会沿着活动链进行处理。当某个活动完成时，会在传送名单上添加完成记录，同时还会加上该活动的补偿（也就是回滚）操作在哪里（例如在某个队列里）的信息，或者会在消息传送名单上给补偿事务注册一个回调。最后它会沿着活动链传递更新的消息。如果某个活动中途失败了，它就会先在本地清理数据回滚，然后将传送名单发往上一个正常完成的活动的补偿地址并执行其对应的补偿事务。上一个活动回滚完之后，也会执行同样的操作，通知再前一个成功活动开始回滚。照此机制运行下去，直到所有之前执行过的事务都回滚完成，保证了整体的一致性。在处理微服务中跨服务边界的事务时，Saga 模式依赖其高容错性和分布式特性很好地取代了传统的事务处理。

Saga 事务模式中的第一个组件是启动器，它会创建事务上下文和交互的原因。然后它会要求一个或多个其他服务（事务的参与者）执行一些业务活动。参与者可以为协调注册并交换消息和请求直到相互之间达成一致或者这些参与的服务已经准备好交互了。相互协调确定好（准备工作）之后，协调者就可以让所有参与的微服务开始逐个提交了。如果任何一个活动完成的时候出错了，你必须执行相应的抵消操作，也称为补偿操作，这个与在常规的 ACID（ACID 指的是原子性、一致性、隔离性和持久性）事务中的回滚是一样的。用代码的语言来描述是这样的一个过程：每个步骤（微服务）执行其自身任务之后，将任务标记为已完成，然后注册其方法，以防接下来的一个活动（微服务）中发生故障，然后将其传送名单交给下一个微服务。如果任何一个地方失败了，那么传送名单会通过调用最后一个成功完成的任务的补偿方法来回传给最后一个成功完成的任务。

举个例子来帮助大家理解。例如，一个用户想要在餐厅预订一张桌子和一些选好的菜品，同时他们还要预订一辆出租车来保证能够按时到达餐厅。这个过程可以分为下面这样几个步骤。

- 检查餐厅中是否有可预订的桌子，如果没有，事务就此终结，不会进行后续步骤。
- 查看想点的菜单项，如果没有用户想吃的菜品，那么也没有必要预订桌子了，也就是说如果这一步失败了，要把上一步一起给回滚了。
- 如果桌子和菜品都订好了，最后需要在指定时间订一辆出租车。如果这段时间订不到出租车，那么用户需要把桌子和菜品的预订都取消掉，也就是说此种情况下，前面两个事务也不会被提交，都必须回滚。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



假设订出租车和餐厅都是在一个网站上完成的。

Saga 模式是一种关注完整性、可用性和质量的架构模式，它也属于服务间通信模式的一种。当考虑 Saga 模式如何落地实现的时候，可能会有一些不同的版本，可以按照自己的方式来实现这个模式中的一些关注点和角色。我们既可以让某一个组件（可以是初始器）来负责整个的控制流，类似编制模式，同时它也可以是一个分布式模式。

还有一个很重要的点是，每个服务的持久化数据是私有的，但是如果采取**一个服务独享一个数据库**的策略，可能某些需要联合不同服务的数据库表的查询是很难做到的。在服务内部做联表查询是可以做到的。在其他情况下，可能需要使用**命令查询职责分离模式**（Command Query Responsibility Segregation；CQRS）来维护一些非规范化的数据视图。

5.4.4 必要时采用混合数据技术

混合持久化（polyglot persistence）是一个用在微服务上下文中的概念。这个词是由 Scott Leberknigh 和 Martin Fowler 提出的。一个企业可以有不同类型的数据模型和不同的访问频率。某些数据会非常频繁地被用到，而且不用过于关心其准确性。另一些数据只在少数非常特定的情况下才会持久存在。具体的情况有很多不同的类型，会对应许多不同的数据。基于这些不同的使用场景，企业会针对不同类型的数据使用不同的技术来存储。例如，会话信息通常会使用 Redis；推荐引擎通常会使用 Neo4j 或者其他的图数据库；对于财务事务管理，通常会使用一些关系型数据库，如 MySQL 或者 Oracle 等。这种情况我们就称之为混合持久化。

电商网站通常要给用户展示一个产品目录。当用户在线选择某一样商品之后，作为一个商人，你想给他们展示一些同类产品，或者是一些可以与选定的商品组合到一起的商品。顾客通常会看到一些这样的消息：“购买 XX 产品的用户都顺便买了 YY 产品。”你会感觉好像有个虚拟的售货员一直在向你兜售一些可以与你买的这个商品配套的东西，直到你付款为止。如果要实现这样一个需求并使用基于 SQL 的关系型数据库的话，那么实现起来是很痛苦的（甚至是不可能的）。另一方面，如果选择的是一个图数据库的话，创建经常一起购买的商品之间的连接是非常简单的，而且还有很多可以改进的空间。所以用户管理部分可以使用基于 SQL 的关系型数据库，推荐引擎可以使用另外一种类型的数据库，如图数据库。

简而言之，某些受益于混合持久化结构的应用给不同的微服务选用了不同类型的数据库。但是这种方式也有其要面临的挑战，例如，在维持数据一致性以及实现不同数据

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

库的查询方面，往往需要团队拥有更深厚的技术能力。

5.5　从单体应用向微服务迁移数据模型

一个典型的企业级应用通常由 3 部分组成：前端通常包含一些 HTML、CSS 和 AngularJS 等 JavaScript 代码；服务器部分用来处理请求和业务逻辑，数据库部分会有一些表结构来存储应用上下文中的各种数据。如果需要在数据库或者服务器端加些新东西，哪怕加的这个东西非常细微，并且与整个应用是隔离的，还是需要重新部署整个应用才能生效。重新部署就会导致死机时间，这种做法在今时今日已经很难被企业接受。

在这些类型的场景中，微服务提供了一种解决方案。但是将单体应用迁移到微服务并不是一件容易的事，需要考虑和决定很多东西。这里我们主要讨论单体架构的数据库模型迁移为基于微服务架构的数据模型的问题。一种迁移办法是一次迈一大步，直接一次性完成整个单体应用向微服务的迁移。另外一种办法是一步一步地迁移，当我们发现一些数据模型单独拿出来也不会影响整个应用的数据模型时，我们开始将这些数据模型抽取出来做成一个单独的微服务。

单体数据库向微服务型数据库迁移时，主要的问题是在提取微服务数据库时经常容易产生一些困惑。没有什么简单计划可以轻松搞定数据库拆分的事。有些既有模式可以帮助我们理解应该用什么样的方式来创建微服务的数据模型。在将单体应用拆成微服务时，应该先搞清楚哪些数据经常一起更新，或者可以找领域专家来解决。下面将会介绍这种领域驱动设计的拆分方式。

5.5.1　领域驱动设计

如果选择给每个服务分配一个数据库，那么怎么来识别各个数据块应该拥有哪些类型的数据呢？这个时候领域驱动设计可以帮助理清思路。这里所使用的数据块这个名词，不完全对应的是一个 MySQL 或者 Oracle 这种物理上的数据库。数据块指的是一些相同类型数据的集合。数据建模和数据应该如何存储在表中是两个不同的问题。数据建模指的是考虑需要使用一些什么样的必要数据来处理实际问题的思考过程。数据建模的主要关注点首先是如何解决业务问题，其次才是持久化相关的技术。DDD 主要也是首先关注业务领域。

为了更好地解释这一点，让我们来看看亚马逊是怎么做的。亚马逊的产品数量繁多，而且相互之间的关联信息也众多，典型的有用户数据、订单详情、配送信息、订单当前位置、售后支持请求等。这些数据存放在成百上千个 PB 中，而且会持续不断地增加。

亚马逊团队在存储这些数据的时候面临了很大的挑战，例如需要保证日常请求都能

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

快速存取到数据、快速加载产品详情等。也许这不算是在解决业务问题。他们所需要解决的业务问题是让世界上任何地方的用户都可以看到任何地方的卖家销售的产品，然后把它们连接起来，向用户展示他们想要的尽可能多的一些选项，然后确保用户坐在家里就能收到自己在网上购买的商品。

假设亚马逊最开始是使用 DDD 的方式来解决这个问题的，那么他们一定会识别出系统中不同的上下文，如用户管理、产品/目录管理、订单管理、报表管理、支付和退款管理、配送管理、卖家管理等。划分了这些不同的上下文之后，就已经将这个大的问题拆分成了一些小的问题。下一步是再来思考这些小的问题，有可能这些第一次拆分之后的问题还可以再进一步拆分成更小的问题。这么一层层地拆下来，就能理解数据是如何从一个组件流转到另外一个组件的了。例如，一个用户可以有两个或者三个地址，所以需要地址管理。顾客可以保存银行卡信息以方便支付，所以在支付这一块可能还需要一个银行卡管理的部分。再下一步是判断哪些数据是与商店相关的，哪些是与其他上下文相关的。例如用户服务中，如果识别出来与用户相关的数据应该在这里，那么这里就是领域和相应的界限上下文。因此，下一步就是在 SQL 或者非 SQL 或者图数据库中存储用户管理数据的问题。这是即将面临的下一个挑战。

DDD 是一个很好的方法，可以按照微服务系统的方式解决业务问题。DDD 的概念其实在微服务提出来之前就已经存在了，DDD 和微服务非常契合。微服务数据模型在 DDD 概念的引导下可以以最好的方式工作。之前已经提到过 DDD 着重点首先在于业务问题。所以使用 DDD 的第一步往往是识别出来问题领域中不同的上下文。领域专家在这个过程可以提供很多帮助，因为他们在这个领域积累了相当多的经验和知识。一旦上下文被识别出来之后，下一步就是定义这些上下文之间的边界。

边界上下文是构成 DDD 的一个重要因素。边界上下文是一种高效地设计微服务边界的方式，有了边界上下文，DDD 技术就可以沿着这些边界上下文的接合处将整个系统划分为一些小的微服务。如果使用 DDD 和边界上下文的方式，可以减少两个微服务需要共享一个数据模型和相应的数据空间或者最终成为紧耦合的可能性。

避免共享数据可以提升我们把每一个微服务看作一个独立部署单元的能力。微服务之间能独立部署之后，就能在保证整个系统仍然安全可用的情况下，有效提升我们的开发速度。DDD 不提倡模型之间共享数据，但是如果仍然要这么做的话，可以使用防护层（anti-courption layer，ACL）的概念，它是用来将数据从一种格式转换成另外一种格式的，还有一些别的好的方式，如事件溯源等，这些我们在后面的篇幅中会讨论。

5.5.2 数据模型迁移方式

前面我们已经提到过，数据模型的迁移可以拆解成一些小的步骤。创建与单体应用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

分离的数据模型的方式有以下几种：

- 视图；
- 使用触发器复制表；
- 事件溯源。

1．视图

数据库视图是数据库按照特定方式排序并展示的一个数据据子集。我们可以控制每个视图显示的行（通过 where 条件选取）和列（select 语句中指定）。视图主要依赖的是数据库技术。开发人员不需要从数据库中提取表，可以创建一个视图然后给视图指定一些代码。视图是一种最简单的方式，而且很多数据库都支持这个功能。视图有个小问题是数据更新时会影响性能，但可以通过使用物化视图来提升一些性能，前提是需要数据库的支持。如图 5-2 所示，可以将一部分代码拆分成一个微服务，然后在数据库中建一个视图。微服务将这个视图作为数据源来获取和更新数据。可以先按照这个设置运行几天看看，从而来进一步验证领域拆分的合理性。

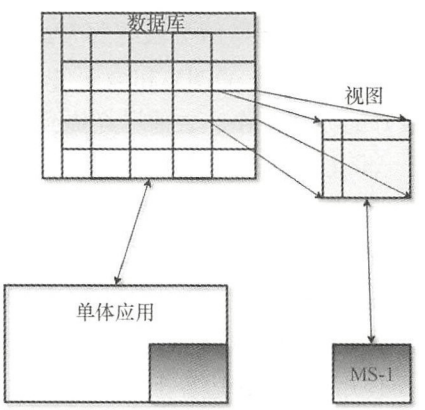


图 5-2

2．使用触发器复制表

另外一种方式是给微服务创建一个新的数据库，然后通过触发器来给新创建的数据库填充数据。这种方式很大程度上还是依赖于数据库的支持。这种方式能提供较强的一致性，但是会需要用到一些诸如 DBlink 之内的工具来让一个数据库可以访问另外一个数据库。使用这种方式的话，有很大概率是最后在各个微服务中使用的都是相同的数据库技术，也就没有享受到混合持久化所带来的好处。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3. 事件溯源

事件溯源是一个功能强大的数据建模方法论，它可以帮助我们避免在微服务间出现共享数据，即使在复杂情况下也能实现。事件溯源是微服务中最好的数据建模方式之一。这里的事件指的是一个发生在过去的活动，它并不是一个状态快照。这个活动本身附带了重演所需要的所有信息。数据库中的数据其实是执行了一系列事件之后所获得的一种状态。所以这里的假设是如果按照同样的顺序执行同样的一组事件，那么数据库的最终状态也会是一样的。使用事件溯源的一个很好的例子是版本控制系统，它会通过一系列的修改历史来保存当前状态。版本管理系统中的当前状态是代码库中最新的源代码，事件对应的就是之前的一些代码提交。

另外一个非常有趣的事实是，这些发生过的事件是不可修改的。事件描述的是一些发生在过去的事，因此无法撤销。这种机制在审计和报表中经常被用到。因为不可修改可以给系统带来最终一致性，所以如果你需要的是一个强一致性的系统，那么事件溯源可能并不适合。用简单的语言描述就是，最终一致性表示系统不能保证在任何时刻都处于一致性状态，系统某些时候提供的信息是过时的，但是，过了一段时间之后，这个信息最终是可以达到一致的。这个时间段可能是 10 秒也可能是 10 小时，完全取决于具体的使用场景和实现方式。

如果你正在从单体应用向微服务应用迁移，并且决定了要用事件溯源的方式，那么我们推荐一个非常不错的工具叫 Debezium。它可以帮你把数据库中发生的改变转换成事件。微服务组件可以阅读和使用这些事件。关于 Debezium 的更多信息，可以在 https://github.com/debezium 找到。

微服务架构取得成功的一个决定因素是，我们在将一个大的问题拆分成小问题的时候做得够不够好。很大可能是我们在将单体应用拆分成微服务的时候会漏掉一些东西，因此黄金法则是不破坏文件，尽量不硬删数据以免日后无法恢复。

4. 示例应用的数据模型

示例应用是一个包含一系列规则的风险评估引擎。基于已有的数据，我们可以执行一个规则然后拿到一个评分。在这个机制下，可以最终确定放款给某个特定用户时会有多大的风险。我们尽力将风险引擎组件概念化，然后定义数据库，并使用一些 DDD 的概念来提出基于微服务的数据模型。

风险引擎需要有用户详情信息。用户可以有很多详情信息，如图 5-3 所示。用户还可以有很多别的数据，但是作为示例应用只列出了这些，目的是基于这些数据使读者理解如何从领域的角度思考数据模型的问题。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

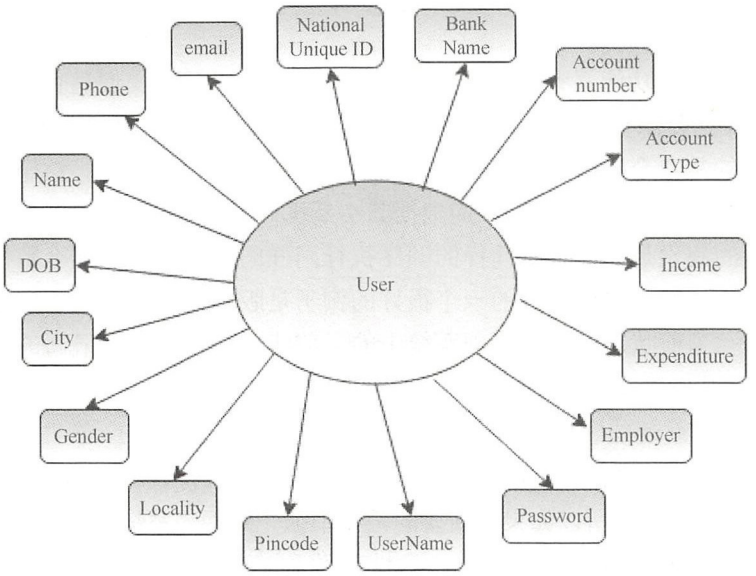


图 5-3

对于上面这个数据模型的例子，我们可以发现所有这些信息都与用户相关。那么问题来了，是不是所有这些信息都应该放在一个领域里，还是可以从这些数据属性中找出几个不同的领域？

事实上，在这些数据中，可以发现 4 个不同的领域或分段。根据数据的相关性和经常一起变更的特性进行划分，如图 5-4 所示。

领域划分比较依赖于经验和专业知识。下面展开讲一下每个领域以及为什么我们要这么划分。

（1）**领域 1（详情）**。这个领域主要处理用户详情。注册用户所需要的最少信息有姓名、性别、出生日期、唯一 ID 和地址等。这些信息可以放在同一个领域，因为同时需要修改这些信息的可能性是很高的。更新这些数据不会影响其他数据模型。例如，如果有人更新了用户的地址，应该是不会影响到银行账号和收入等数据的。

（2）**领域 2（财务）**。领域 2 主要包含财务信息，可能有用户银行账号详情和收入信息等。这些类型的信息是有一定关联的，例如，如果用户是一个按月领取薪水的雇员，当他更换工作的时候，他的收入和银行账号有很大的可能性都会发生改变。因为有些公司是在合作银行开立收入账户的，因此换工作到这些公司的时候，很可能需要用一个新的合作银行的账号来发工资。

新公司可能会要求用户在它们的合作银行开一个新的账号。这也是我们会把这些信息放在一起的原因。比较有趣的是，雇主变更的时候也有可能需要更新用户地址信息，

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

那么是不是还要把用户地址也放在这个领域呢？这主要取决于过往的经验和所掌握的目标用户的数据集。通常这是领域专家应该派上用场的时候。例如，专家可能会按这种方式考虑：经验丰富、技术全面的用户绝大部分都买过房子，也有家庭了，这些用户不会经常变更地址信息。因此需要通过数据集，找出数据中一些经常同时修改的模式。

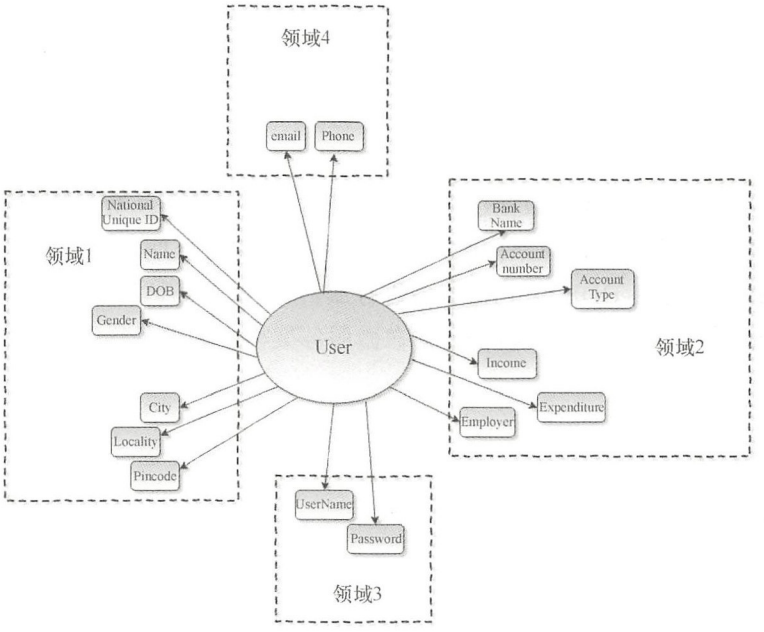


图 5-4

拿风险引擎应用来说，假设银行试图提供高额贷款。出于风险考虑，一般来说只有用户收入水平较高或者已经买了房子之后，才会发放高额贷款。因此，高额贷款对应的用户不太有可能会经常变更地址。所以有了这些认识之后，地址和收入数据没有被放在同一个领域中。在具体操作中，需要平衡领域和数据集模式才能划分好合理的领域。

（3）**领域 3（认证）**。领域 3 是一个简单的认证领域。应用的认证和用户授权方式是一块比较特殊的业务。这也是把认证信息一起放在这个单独的领域中的原因。

（4）**领域 4（通信）**。领域 4 主要处理应用与用户通信的渠道。从数据模型中可以看出，电话号码和电子邮件会被用来与用户通信，这些信息都适合放在通信领域中。如果有移动端应用，那么在做手机推送的时候需要使用到移动设备的 ID，这些信息也可以放在通信这个领域中。

接下来便可以开始考虑数据的存储问题了。所有这些数据都可以使用关系型数据库来存储，例如，可以给每个领域分配不同的数据库，对应的每个数据库中会有一些表来

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

存储相关信息。领域之间通过用户 ID 进行关联。当新用户注册的时候，他的信息会作为一个创建事件和用户 ID 一起发布。所有其他服务会订阅这个事件，在监听到创建用户事件之后，可以在各自的数据库中创建相应的数据。生成用户 ID 的职责可以交由第一个详情组件来完成，这个生成的用户 ID 在所有其他组件中都会用到。

这是一种为应用建模的方式。现在你应该理解从领域和数据集的角度开始划分系统的思路了，而不会一上来就先构思关系型数据和存储的问题。

5.6　小结

本章讨论了微服务时代的数据建模问题，还讨论了一些传统数据建模中的常见问题在微服务环境中的表现形式。本章同时介绍了一种保证运行长期事务的模式，也就是 Saga 模式，还涉及如何使用 DDD 来为微服务进行服务划分。最后以一个微服务的例子讲述了整个应用的建模过程。下一章将会解释在微服务架构中如何进行相关测试。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 6 章
测试微服务

平台中微服务数量的增加将导致 DevOps 的工作量呈指数级增长。要想应对这一挑战，首先必须实现工作流程自动化，这为自动化测试带来了挑战。然而，测试微服务并不像测试单体应用那样简单。由于微服务之间可彼此调用 API，因此在孤立环境中对微服务开展集成测试将是非常复杂的工作，测试规划需要付出极大的努力。本章将讨论一些微服务测试技术。

本章将会覆盖下列主题：

- 微服务中测试的目的；
- 单元测试；
- 集成测试；
- 组件（服务）测试；
- 契约测试；
- 端到端测试。

6.1 微服务中测试的目的

微服务在许多方面均有别于单体应用，所以二者不能采用相同的测试方法。对微服务而言，更高的发布频率使发布速度成为关键因素。为了确保快速发布，微服务的测试也应该快速完成，但不能因为追求速度而忽略质量。对微服务逐一开展单独测试，如单元测试，虽然有些烦琐，却是很好的实践。然而这种测试方法并不总是可行的。因为微服务有时会与第三方应用进行通信，有时会与其他微服务通信，所以仅开展单元测试并不足以保证质量。例如，微服务 A 发生任何变化都有可能影响微服务 A 与微服务 B 之间的通信。开发人员和质量保证人员必须确保在整个开发过程中不对任何组件造成破坏。一般来说，虽然各个团队负责开发多个不同的组件，而这些组件会对微服务之间的通信产生影响，因此开展契约测试是非常有必要的。此外，质量保证人员还应对整个微服务

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

开展手动或者自动的端到端测试。

综上所述，显然没有任何一种方法足以确保新开发的微服务能够同时满足兼容性、完整性和质量要求。我们可以结合使用多个最适合的测试策略来实现这些目标。图 6-1 所示的层级结构图的架构并非是固定的，也并非适用于所有组织，却可以作为微服务测试的范例模型供参考使用。

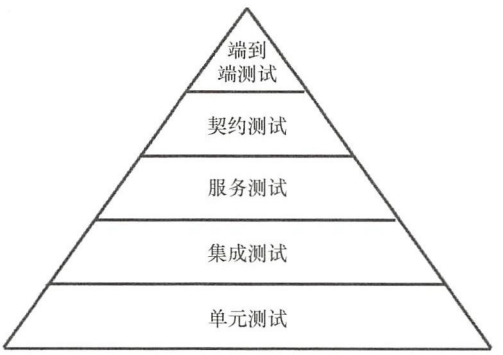


图 6-1

每个组织均可根据自身的团队情况和需求来调整这个层级结构。如图 6-1 所示，微服务测试始于单元测试，止于端到端测试（end-to-end，E2E）。我们将在下面几节中对各级别的测试进行逐一讨论。

6.2　单元测试

单元测试由来已久，其测试原理是使用代码来验证假设。即使代码在一段时间之后发生了改变，其假设应该依然有效。由于此类测试是针对特定的代码单元而开展的特定的假设测试，所以被称为单元测试。假设在一个 Java 类中有 10 个方法，并且每个方法都有 10 种不同的用途。那么对单元测试而言，则需要测试 10 个代码单元。例如，其中一个方法是将 ID 作为输入，然后返回一个布尔值来表示该 ID 在系统中是被启用还是被禁用。下面是针对用于检查指定 ID 或者说用户 ID 在系统中是被启用还是被禁用的方法的示例代码。

```
public boolean isEnable(String Id){
    return someRepo.findOne(Id).getStatus().equals("ACTIVE");
}
```

上述方法应该返回 `true` 或 `false`。但是，如果此 ID 不存在于数据库中，那么程

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

序的行为将会是什么样子？是应该抛出异常还是返回一个值？这种情况下，开发人员将不得不回过头去再次查看相关假设。如果该假设指示应返回 false，那么开发人员将必须更改代码。这就是单元测试的过程。

下面让我们以最常见的计算器为例，对单元测试进行讨论。计算器对两个整数进行算术运算，代码如下：

```
public class Calculator {
    public static int addition(int number1, int number2) {
        return number1 + number2;
    }
    public static int subtraction(int number1, int number2) {
        return number1 - number2;
    }
    public static int multiply(int number1, int number2) {
        return number1 * number2;
    }
    public static int divideInteger(int number1, int number2) {
        if (number2 == 0) {
            throw new IllegalArgumentException("Cannot divide by 0!");
    }
        return number1 / number2;
    }
}
```

下面是面向上述计算器代码的一些单元测试用例。

```
public class ExampleCalcTest {
    @Test
    public void shouldPass_forAdditionOfNumber() {
        assertEquals("error..", 8, Calculator.addition(1, 7));
        assertEquals("error..", 23, Calculator.addition(17, 6));
        assertEquals("error..", 17, Calculator.addition(8,9,));
    }
    @Test
    public void shouldFail_forAdditionOfNumber() {
        // assertNotEquals(String message, long expected, long actual)
        assertNotEquals("error..", 0, Calculator.addition(13, 2));
    }
    @Test
    public void shouldPass_forSubstractionOfNumber() {
        assertEquals("error..", 5, Calculator.subtraction(12, 7));
        assertEquals("error..", 3, Calculator.subtraction(4, 1));
    }
    @Test
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    public void shouldFail_forSubstractionOfNumber() {
        assertNotEquals("error..", 0, Calculator.subtraction(2, 1));
    }
}
```

测试驱动开发（TDD）倡导单元测试先行。这些单元测试用例只是为了验证微服务中特定代码单元内部的逻辑。微服务内部的类和方法的多少，决定了单元测试用例的数量。测试用例可以自动运行，例如，在构建基于 Java+Maven 的工程时，单元测试可在工程进行构建时自动运行。Jenkins 构建工具将从源代码版本控制系统中（如 Git）获取最新代码，接下来基于这个最新提取的代码运行单元测试用例。待这些测试用例成功完成之后，进行工程构建并上传至二进制代码库中，同时向 Git 代码库提交新的标签。如果单元测试用例失败，Jenkins 构建工具会将构建标记为失败，并向相关人员反馈。图 6-2 描述了上述几步操作。

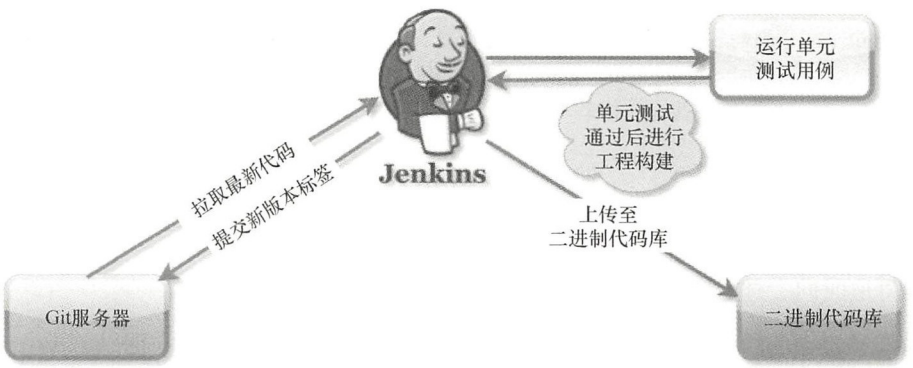


图 6-2

单元测试用例只覆盖了一小部分代码。因此随着代码量的增加，单元测试用例的数量也会随之增加。因为单元测试只针对一小段代码进行测试，所以当被测应用运行单元测试用例出现问题时，可以很方便地确认问题的确切原因。因为无论哪个测试用例失败，只需检测与之相对应的一小段代码即可，在一小段代码中确认问题要快很多。

对一些有外部依赖的特定代码进行单元测试时，则需要由**模拟服务器**（mock server）来隔离依赖关系，即对于给定的输入，模拟服务器可以生成预期的输出结果。依赖关系的模拟应遵循一定的规则，例如，对于给定的 *Y* 输入应生成 *X* 输出，以便使测试用例编写人员能够专注于编写测试用例用到的相关代码。到目前为止，我们只讨论了如何在隔离情况下来单独测试小块代码，前提条件是其他代码单元都能很好地工作。接下来，我们将会讨论如何使用集成测试来测试这些不同的代码单元协同工作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

6.3 集成测试

单元测试用例进行测试检查与代码验证，但只会验证这段代码的内部逻辑，并不验证其依赖的模块。假设服务层 A 的方法依赖于服务层 B 的方法，当我们为服务 A 编写单元测试用例时，将需要由模拟服务器来模拟服务 B。这种情况下，如果服务层 A 和 B 均通过了所有的单元测试，则说明对于服务层 A 和 B 的单元测试用例已经分别通过，但并不能说明服务层 A 和 B 的交互就是正常的。尽管服务层的验证需要独立进行，但在微服务环境中，不同服务层的不同方法之间的通信情况也需要进行相关测试。

因此需要引入集成测试，以确保来自不同服务层的组件均可完美协作并提供预期的结果。例如，数据层从数据库中获取数据，服务层调用数据库层并将数据转换为所需的格式发回给用户。在理想情况下，服务层拥有自己的单元测试用例，然而因为服务层依赖于数据层，所以在编写服务层的测试用例时，数据层的调用及返回结果会由模拟服务器进行依赖隔离。由于服务层的测试用例只关注服务层代码，这有可能导致服务层发送给数据层的输入实际上并不能被数据层所接收。个中原因多种多样，而且，与服务层的逻辑毫无关系，如数据层的有效性验证失败等。但我们已将数据层的不同返回值模拟到服务层的测试用例中，因此，由模拟服务器模拟的数据层将会接受服务层发送的值并做出预定义的响应，从而使得服务层的测试用例执行成功。但在实际测试中，由于服务层发出的值可能并不被数据层方法所接受，这会导致测试失败。

微服务架构中可能包括许多外部服务调用，那么是否需要对外部服务（服务代码的外部）进行测试呢？答案是不需要，因为在集成测试阶段，应由模拟服务器来模拟外部服务，开发人员无须对外部人员的代码负责。有些人可能持反对意见，认为应将外部服务测试纳入到集成测试之中，但作者认为外部服务测试应该隶属于端到端测试。这一点将稍后进行讨论。

在集成测试中，测试人员或开发人员负责测试整个子系统，而不仅仅是一小段代码。集成测试针对的是整个模块。集成测试试图分析整个子系统的行为，因而集成测试用例的影响范围比单元测试用例更大。集成测试与微服务中对等的方法或服务层进行交互，以确保它们以期望的方式进行通信。

6.4 组件（服务）测试

在微服务的上下文中，任何服务本身均可被视为一个组件。因此，整个微服务的测

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

试可以称为组件测试。顾名思义，组件是指被合理封装在更大规模系统之中、与该系统一致的并且可以被独立替换的部分。

任何已部署完毕、随时准备投入生产使用的微服务都必须接受服务或组件测试。所以此服务或组件绝对不能具有特定的测试逻辑。

服务测试或组件测试将涉及真正的网络调用或数据库调用，而数据库测试需要测试人员执行配置注入操作，由于测试人员需要启动和停止测试数据库并且进行种类繁多的其他配置，这也加重了测试人员的工作负担。由于此类测试运行在类生产环境中，因此还会遇到一些没有必要在现阶段进行处理的实际问题，例如，对数据库的操作可能延长了测试计划的执行时间，或者网络调用也耗费了测试时间，有时甚至会丢包。

应对这些挑战的方法之一便是复制一个原始外部服务来专门为测试所使用。那么现在的问题是这些测试用的副本服务是应该通过测试环境注入还是应该通过代码加载进 RAM。一种解决方法是通过测试的相关配置来进行控制。对数据库而言，如果使用了基于关键字的数据库分片技术，那么，通过使用关键字 `I'd like TEST` 可以将测试数据存储在不同的数据库中。这样，在对生产环境做测试时，生产数据与测试数据不会混合存储在同一个数据库中。尽管如此，也不建议直接在生产环境中测试。但在某些情况下，在生产环境中进行健全测试是在所难免的。这时可能会产生很多疑问，例如，测试过程中可以使用内存数据库吗？这种方法会引发真正的网络调用吗？在生产环境中测试会有代码漏洞吗？等等。

然而主要问题还是没有解决，即调用数据库或任何第三方应用都是非常耗时的工作。假设我们可以在内存中创建这些测试用的副本服务和数据库。这种做法的最大好处便是能够避免网络调用并缩短测试用例的运行时间。想象一下，对于数以千计的测试用例或 Selenium UI 测试用例，根据其数量和复杂程度不同，可能需要运行 5～6 小时。此外，如果这些测试用例通过网络进行通信并在数据库中添加数据，运行时间将会更长。下面这种做法可加快测试用例的运行速度并降低组件的构建复杂性。为减少网络调用带来的麻烦，可以考虑使用预构建的库，例如使用面向 JVM 的 inproctester 和面向.NET 的 plasma，从而使组件测试更贴近真实情况。数据库亦是如此，例如，使用像 H2 这样的内存数据库，或者考虑使用内嵌 Elasticsearch 和 Neo4j 等工具的数据库。

做好上述所有的工作，就可以在孤立的环境中开展微服务组件测试，还可以使我们对测试环境拥有更多的控制能力，并且比较容易地复现实际生产环境中出现的问题。例如，对于某类型的请求的错误，开发人员可以修改代码，并将生产环境中的数据复制到测试环境中，并把这类请求指向内存中的测试服务和内存数据库，达到方便调试的目的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

6.5 契约测试

契约是解释双方交易方式的协议。同理，微服务上下文中的契约是指服务被调用时协商好的调用格式。或者说，在微服务的上下文中，微服务契约是指消费者和 API 提供方之间对确切通信方式的描述以及对期望的输入和输出格式而达成的协议。契约测试有别于集成测试。例如，对一个贷款服务应用而言，在批准客户贷款申请之前，该应用将调用第三方服务，以了解此客户可能在别的银行或借贷公司的贷款金额。假设由于某种原因，第三方服务一方的契约被更改，那么应用的集成测试仍然会正常运行，因为我们所使用的模拟服务模拟了既定的返回值。但在真实的生产环境下，这个调用将会失败。这就是契约测试产生的原因。

所有的服务均应逐一单独调用，且调用返回的结果应与契约相匹配。服务调用结果应只匹配输入和输出契约，无须考虑内部代码。与其他测试技术一样，契约测试也应模拟任何依赖的服务。对契约测试而言，测试人员必须为 API 提供方设计并运行测试用例，之后将测试结果存储起来。契约测试可通过两种方法进行。第一种是在客户方/消费者运行相关测试用例，然后将传递给提供方服务的输入值和来自消费者服务的输出值进行匹配。第二种方法正好相反，首先在客户端运行测试用例并保存其输出值，然后将这个输出值作为输入值传递给服务提供方。接下来，以相同方式比较预期结果和实际结果，确保未出现违反契约的情况。

在接口、请求参数和响应参数的开发过程中开展契约测试，对提高开发准确性帮助很大。通过这种方法开发出的服务将会异常健壮，始终能给消费者带来预期结果。例如，如果将 API 提供给其他服务，那么，开发人员便应考虑契约测试，这可以保证在 API 添加新参数或更改功能的情况下的可扩展性，以确保服务之间的契约不会因此而违约（除非所有服务均发生了重大改变）。即使发生了这样的改变，契约测试也能自动保证契约不会因此而违约。**Pact** 和 Spring Cloud Contract 能帮助开发人员达成这些目标，这是两种开源工具，可以在 GitHub 上下载。

6.5.1 Pact

消费者驱动的契约测试日益引起微服务架构开发人员的关注。考虑到彼此通信的服务数量如此众多，微服务中的任何一个地方发生细微变化就可能违反契约。解决这个问题需要借助契约测试和自动化工具。Pact 就是为此而生的，可访问 GitHub 代码库 https://github.com/realestate-com-au/pact 查看详细介绍。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Pact 的工作原理很简单：对 API 提供方开展测试，模拟其所有依赖关系并保存测试应答。之后，重放此类应答来响应消费者代码，则应获得与 API 提供方操作相匹配的输出结果。Pact 有着基于不同语言的实现，例如，与 JVM 相关的 Pact JVM，也有基于.NET、Ruby、Python 和 Swift 等语言的实现。

6.5.2　Spring Cloud Contract

对于消费者驱动的契约，Spring 还提供其他项目来帮助用户轻松完成契约测试，即 SCC（Spring Cloud Contract）。SCC 主要由以下 3 个工具组成：

- SCC verifier tool；
- SCC WireMock；
- SCC Stub Runner。

默认情况下，SCC 支持基于 JSON 的桩模块（WireMock）。SCC verifier tool 应添加到服务提供方。我们必须为微服务的不同请求定义契约，然后将其保存在与项目工程文件相同或不同的代码库中。这个代码库作为共同契约代码库，将被服务生产者和服务消费者同时使用。下面是 POM 示例文件。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-dependencies</artifactId>
            <version>1.1.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-contract-verifier</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-maven-plugin</artifactId>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                <version>1.1.1.RELEASE</version>
                    <extensions>true</extensions>
                <configuration>
                    <baseClassForTests>in.packt.example</baseClassForTests>
                </configuration>
            </plugin>
        <plugins>
    </build>
```

对于 baseClassForTest 的配置部分，也可以使用 regEx 进行正则匹配。

这个想法在 Spring 契约中实践起来非常简单：创建一个契约和一个端点（endpoint），运行测试并自动生成 stub.jar 文件。我们可以在本地 Tomcat 中运行 stub.jar 文件，消费者一端可调用这个文件，并对比测试结果，从而捕获到集成测试或之前测试未发现的问题。因此，契约测试能降低系统在部署到生产系统时的整体风险。

6.6 端到端测试

端到端测试主要是指测试整个应用，包括控制器层、服务层、数据库以及任何第三方依赖。端到端测试旨在确保系统中的每个组件都能以它们应有的方式良好运行。端到端测试不应去模拟任何依赖关系，而应该站在最终用户的角度来测试，因为是用户使用整个应用。

在单体应用中，端到端测试力求覆盖应用中所有可能出现的场景。然而“应用”一词在微服务情境下已经发生了巨大变化。在微服务架构中，每项微服务本身就是一个小而独立的应用。由于微服务的各部分分散在各处，因此测试整个应用变得非常有挑战。例如，在微服务应用中，端到端测试中出现了很多问题，如要测什么、测多少以及测试范围等。端到端测试可确保每个组件都能按照预期的方式和要求协同工作。

有些测试框架可帮助实现端到端测试的目标。**JBehave** 就是可以进行自动化功能测试的工具之一。它采用从 TDD 演进而来的**行为驱动开发**（behaviour-driven development，BDD）方式。JBehave 可将用户故事作为输入并确保系统的响应符合我们的预期。此类测试的规模和复杂性都远远高于其他类型的测试。

根据测试金字塔，用户界面测试是最高级别的测试，因为它从最终用户的使用角度出发对系统进行测试。用户界面测试必须感觉如同用户在与系统互动。所有的数据库、不同的接口以及内部和第三方服务等均须无缝协作才能得到预期的结果。

6.7 更进一步

在微服务领域中，端到端测试不应仅限于需求测试，还要进一步考虑性能测试。在

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务架构中，每项服务均运行在不同的机器上，被其他服务调用的所有服务都会有一个端点。因此，在对这些端点开展性能测试时，是应该开展包括端点的隔离的组件级测试还是开展应用级测试呢？

如果对组件级进行负载/性能测试，那么识别出哪项服务运行缓慢将会非常重要。此外，不同层的问题、网络层的问题和机器级别的问题都应是此类测试的关注重点。此类测试的目的是发现性能瓶颈，同时能够帮助识别被测应用在网络延迟、数据丢包和组件扩展策略方面的各项表现。性能测试对测试人员来说是一项艰巨任务。任何新版本的推出都可能会导致用户界面层性能变差，从而使识别哪个微服务变慢成了一项挑战。这时便需借助在微服务架构中扮演重要角色的监控技术。我们将在第 9 章中讨论如何监控微服务。

6.8　小结

微服务测试本身是一项极具挑战性的任务，没有什么银弹可供使用。微服务测试取决于团队结构、DevOps 实践能力以及可用的基础架构等多种因素，这些因素决定了不同类型测试在规则、流程或顺序上的变化。基于组织的需求，结合使用多项测试技术是最有效的测试方法。自动化测试在微服务架构中非常重要，是加快微服务测试速度的另一个驱动轮。自动化测试将会加大 QA 团队的工作负担，因为他们需要编写并维护自动化测试用例。然而需要注意的是自动化测试并不能完全取代手动测试。对于实际用户体验或情感测试等项目，手动测试的帮助更大。因此联合开展这两类测试才能更富有成效。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 7 章
部署微服务

微服务架构的优势在于它能把一个大问题分解成多个小问题，然后分别处理这些小问题并独立发布这些代码，从而降低出错的波及范围，因而能够避免发生系统级别的宕机。批量大小（Batch Size），即微服务的发布速度，是微服务架构中的另一个重要维度。微服务的发布周期可以是 1 个月、2 个月或 6 个月不等。传统发布方法因发布频率较低而导致开发人员不太熟悉发布流程，因此更容易犯错。此外，传统发布方法还涉及许多手动操作，如手动关机、安装/更新基础架构、部署、重启和手动测试等。手动操作步骤越多，出错概率越高，整个发布过程越是烦琐、冗长而耗时。

如果两次发布间隔过长，那么，不同组件之间可能会因为版本冲突和组件不兼容等问题而无法协作，而我们也会因为期间发生太多变化而难以找出问题是由哪些变化引起的，经常会陷入“亡羊补牢，为时已晚”的困境，也无法及时获得反馈。

相比之下，亚马逊等厂商的发布频率则高达每周一次甚至一天多次。由于发布频率会影响停机时间并缩小维护窗口，所以为了在维护窗口与批量大小之间保持平衡，我们必须在零宕机的情况下发布产品。

本章将会覆盖下列主题：

- 持续集成；
- 持续交付；
- 用微服务配置 CI 和 CD 工具；
- 微服务的 Docker 化；
- 微服务的流水线。

7.1 持续集成

业务需求总是瞬息万变。为了保持足够的竞争力，企业总是从不同的视角来审察数据、市场和趋势等因素，并努力与最流行的趋势保持一致。商业人士喜欢分析形势，乐

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

于尝试新鲜事物，热衷于从分析结果中学习，从而尽可能快速地增加业务价值。

这些变幻无穷的业务需求必须在系统中快速推进，以尽快评估其潜在的影响。传统的软件开发周期漫长且乏味，乃是引发问题的核心。周期越长，问题越多。每名开发人员都从代码库中拿到一份代码副本。所有开发人员的起点都是相同的，并且都在添加新功能。工作开始之后，所有的开发人员都在不断地更改和添加代码。每个人都在为完成自己的任务而努力。在整个工作团队完成任务后，开发人员纷纷将代码推送到中央代码库中，那么问题由此出现。中央代码库中的代码或某些方法签名可能已被改动，或者某些 API 可能已发生改变，或者来自另一个团队成员的未知更改导致测试开始失败，有可能出现代码合并冲突、编译冲突和测试冲突等诸多问题。在这之后，又要开启另一个周期来修复这些问题，这将会延长发布周期。不仅如此，新产品发布之前还需开展有关系统整体性能的测试，以确保软件能够高效工作。这将涉及手动 QA 测试、缓慢的版本调整过程以及手动性能测试等太多的测试工作。

那么，如何做到快速呢？让我们假设另外一种场景。开发人员在本地更改代码，在本地开展单元测试。开发人员提交代码，由**持续集成**（continous integration，CI）系统检查代码并对其运行全套的测试用例，以确认新提交的代码不会破坏构建过程并确保不存在编译时问题。这可能需要每天开展多达 1000 次测试。当所有的测试都成功执行后，成功的构建将自动部署至预生产环境，准备接受验收测试。在成功通过验收测试后，预生产环境中的构建便可随时正式发布到生产环境。

持续集成是指每一次代码库有变更后，尽早开展自动化测试。有时候，想法越简单，实践起来越困难。

持续集成的实践需要借助一些出色的 UI 测试工具和后端测试工具，如下面列出的几个 UI 测试工具/框架：

- Jasmine（JavaScript）；
- Selenium（UI 测试）；
- CasperJs（UI 测试）；
- Cucumber 或 Lettuce。

每一次代码改动后就马上运行所有测试的好处在于能及时发现此次改动是否破坏了某些功能。任何失败都可以被马上修复。如果我们决定持续测试每一次变更，则应开展自动化测试。持续集成是实现持续部署的第一步。

7.2　持续交付

提高发布频率可以降低发布过程中的出错风险。提高发布频率的唯一办法是自动运

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

行整个发布流程，包括打包发布、基础架构设置、部署和最终的可用性测试。我们要尽可能减少手动操作以增加发布频率。这个阶段称为**持续交付**（continuous delivery，CD）。维基百科对其定义如下：*持续交付是工程团队在较短周期内制作完成软件的一种软件工程方法，以确保随时可靠地发布软件。*

持续交付是使得任何组织尽可能快速高效地向用户交付新功能的软件策略。持续交付的核心是创建一个可重复的、可靠的、逐步改进的流程，从而实现软件从开发者到客户的实时快速交付。我们通过确保代码始终处于可部署状态来实现持续交付，即使成千上万的开发人员每天都在更新代码也不例外。这也正是持续集成通常与持续交付相伴随的原因。持续集成是实现持续交付的前提条件。

持续交付借助于持续交付流水线来实现软件生产线自动化，从而持续调整软件的生产流程。有时候，频繁发布软件会让人们担心软件版本不够稳定。但实际情况并非如此。研究表明，快速交付的团队基本上都是高效的团队，可以从始至终高质量地交付软件。相反，不能快速交付的团队往往效率不高，从而影响软件交付质量。持续交付没什么魔力，只需要坚持不懈地持续改进并始终保持团队的高效合作即可。

图 7-1 明确表示出了持续交付的概念。开发人员开发出某项功能，然后将其部署至预生产环境，进行相关测试并最终发布。持续交付中，从测试到发布的过程是一个手动过程。例如，在测试完成后，QA 团队通常会人工检查测试结果，确保测试没有问题后，再将软件发布到生产环境中。因此，持续交付中仍存在手动认可环节。持续交付允许开发人员自动运行所有的测试用例，以便他们能够在部署至生产环境之前，对应用程序开展多维验证，以确保最终发布顺利进行。

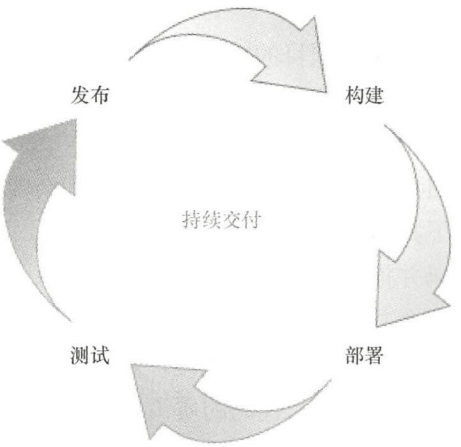


图 7-1

如果我们将验收测试环节也变为自动化流程，那么，便会衍生出另一个 CD，即**持**

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

续部署（continuous deployment）。这两个 CD 之间的基本区别如图 7-2 所示。

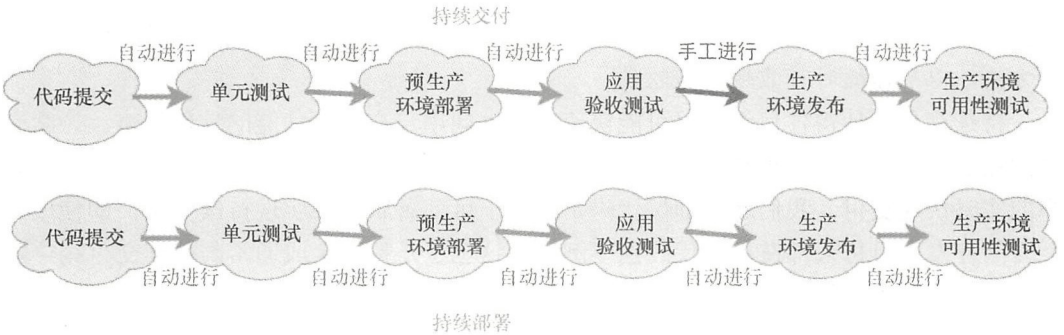


图 7-2

持续交付过程中的手工干预操作，在持续部署中变成了自动化操作。稳定的持续交付是实现持续部署的前提条件，持续部署是在稳定的持续交付基础上开展的下一步行动。自动化是实现任持续交付或持续部署的必经之路。代码提交后自动触发构建、自动化测试和频繁部署等，都是实现持续部署的必需步骤。持续部署是每个技术部门的最终目标，但业务部门也需为此做好准备。总之，通过定义持续集成和持续交付的实践方法并了解二者的作用领域和优势，相关工作团队能够更加快速高效地创建任务，并区分优先级后更合理地完成相关任务。

7.3　用微服务配置 CI 和 CD 工具

构建、测试和部署是任何 Web 项目的完整组成部分。CI 和 CD 工具可用于自动执行这 3 步操作。如果想找出适合某工程项目的 CI 和 CD 工具，要先确保该项目支持前面的 3 类自动化操作。

图 7-3 解释了软件项目的通用工作流程。该流程可能会多种多样，但其核心工作流程不变。让我们来了解一下具体步骤。

（1）开发人员开发代码并推送至代码库。

（2）开发人员发出代码合并请求，以便其他组员能够评审所提交的代码。

（3）其他组员评审这名开发人员提交的代码，基于评审结果，批准此次提交或提供一些改进意见。基于合并请求的大小和代码的质量，评审活动可能会重复 2～3 次。

（4）评审者将代码合并到分支中，这将自动触发软件包构建流程。

（5）构建好的软件包将被自动部署到某个环境中，即我们所说的开发环境。

（6）所有种类的测试用例均应该在新的构建中自动执行。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

（7）给出集成测试报告。如果测试报告标记为成功，则证明这个构建已经可以部署到生产环境中。

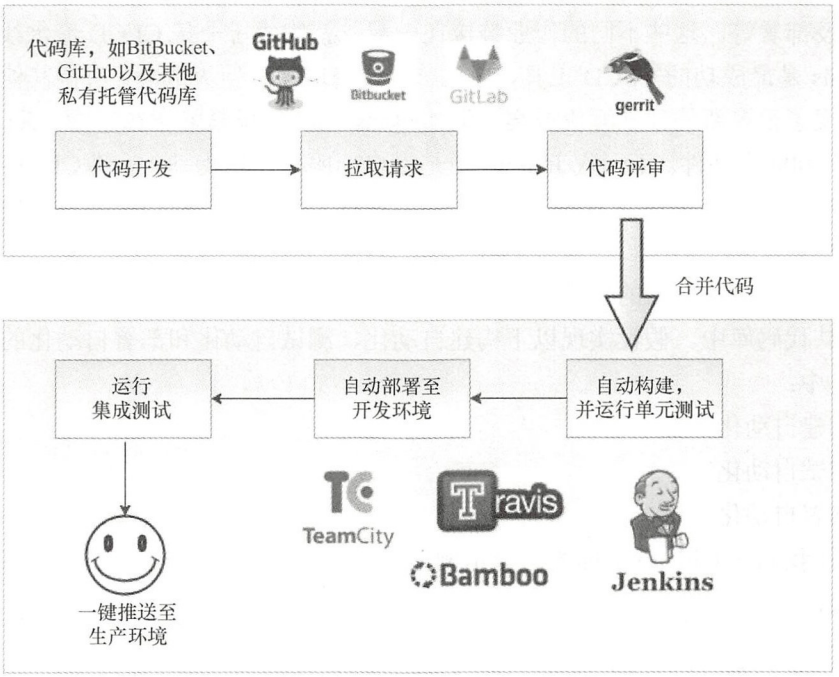


图 7-3

不同组件每天可能会接受多达 100 次测试，这让 CI 和 CD 的存在更有意义。

下面是市面上可用的 CI 和 CD 工具：

- Jenkins；
- TeamCity；
- Bamboo；
- GitLab（托管）；
- Travis（托管）；
- Codeship（托管）；
- CruiseControl（托管）；
- CircleCI（托管）。

这些工具以及其他 CI 和 CD 工具大致可分为两类：

- 独立软件解决方案；
- 托管解决方案。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Jenkins 或 Team City 等独立软件的解决方案可安装在服务器甚至笔记本电脑上，而 Codeship 及 Travis 等托管解决方案则像软件主机一样安装在其他地方，我们需要通过凭证在浏览器中访问它们。这些软件最终都是帮助我们创建可执行某些操作的作业，如构建、测试及部署等。这些不同的作业链接在一起，就构成了一条 CI/CD 流水线。

Jenkins 是最成功的 CI/CD 工具，它从一开始便取得了巨大成功，引得其他厂商纷纷效仿，涌现了很多新的独立解决方案。但 Jenkins 已将市场紧紧握在手中。Jenkins 现在拥有超过 1 000 个插件。下一代 Jenkins 更是以自动服务发现为基础，将 CI/CD 工具推向云端，其中大部分 YML 配置都与 CI/CD 流水线之中的代码共存在同一个代码库（如 Bitbucket）中。

示例中的微服务使用的便是 Jenkins。我们假定代码能够托管在 GitHub 或 Bitbucket 等任何公共代码库中。假设实现以下构建自动化、测试自动化和部署自动化的示例托管在 GitHub 中：

- 构建自动化；
- 测试自动化；
- 部署自动化。

首先让我们在 GitHub 上面为这个示例创建一个名为 `CI_CD_sample` 的代码库，如图 7-4 所示。

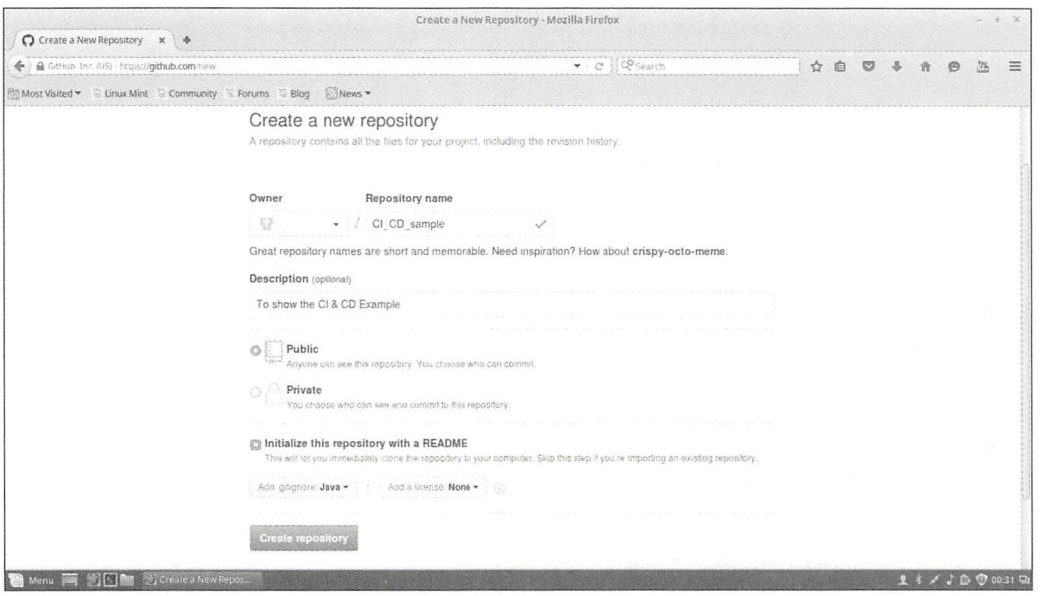


图 7-4

代码库设置为 Public，并添加 `README` 文件和基于 Java 语言工程的 gitignore 文件。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

然后将该工程复制到本地并开始创建一个基于 Spring Boot 的微服务样例。为了使用 Maven 作为构建脚本，需要在该工程的根目录中创建一个 POM 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>CI_CD_sample</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

</project>
```

返回 HelloController 的示例代码如下：

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/greet")
    public String greeting(@RequestParam(value="name", required=false,
    defaultValue="World") String name, Model model){
        return "Hello "+name+"!";
    }

    public static void main(String[] args) throws Exception{
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        SpringApplication.run(Example.class, args);
    }
}
```

首先运行下面的命令来运行这个示例：

```
$ mvn spring-boot:run
```

然后使用下面的命令将代码推送至 GitHub：

```
git add .
git commit -m 'Added sample microservice'
git push origin master
```

对于 CI/CD 服务器，既可以使用托管解决方案，也可使用独立软件解决方案。本书以在 Linux 环境中安装 Jenkins 为例，使用下面的命令完成安装（根据 Jenkins 的 wiki 页面说明）：

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key |
sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/Jenkins.list'
sudo apt-get update
sudo apt-get install Jenkins
```

安装完成后，在浏览器中访问 http://localhost:8080，即可看到图 7-5 所示的界面。

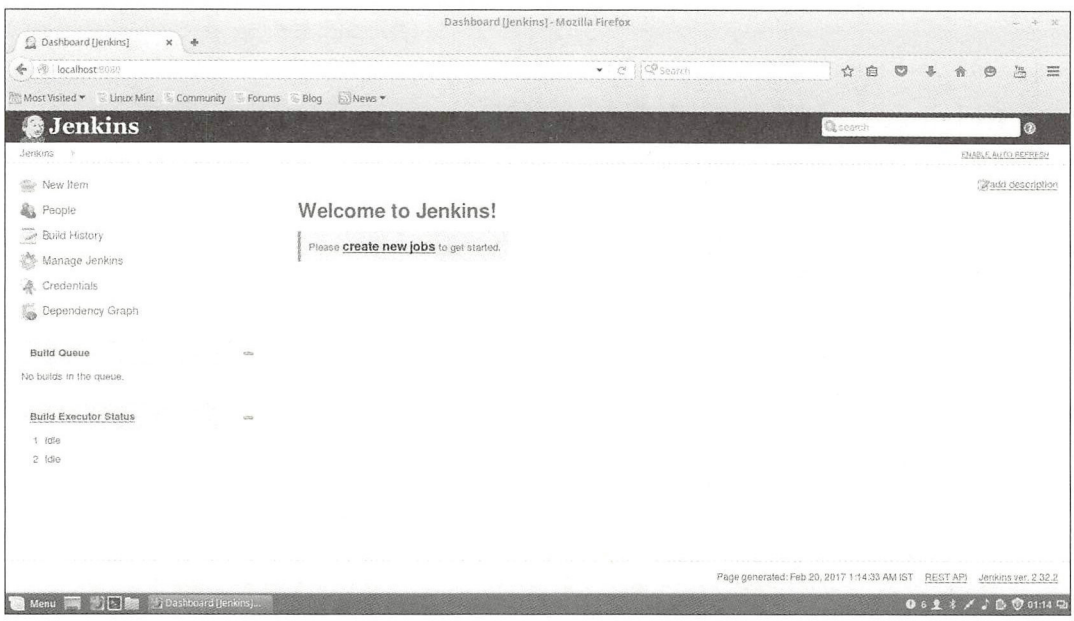


图 7-5

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

为了与 GitHub 集成，需要将 Jenkins 服务器密钥作为部署密钥添加到 GitHub 中，并向 Jenkins 添加新插件。在 **Manage Jenkins | Manage Plugins** 中选择并安装 GitHub 和 Git 插件，并重启 Jenkins 以完成安装。

安装 Maven 插件。

单击 **Manage Jenkins | Global Tool Configuration | Maven installations**，并指定 Maven 的安装位置。

下一步是创建 Maven 构建作业，如图 7-6 所示。

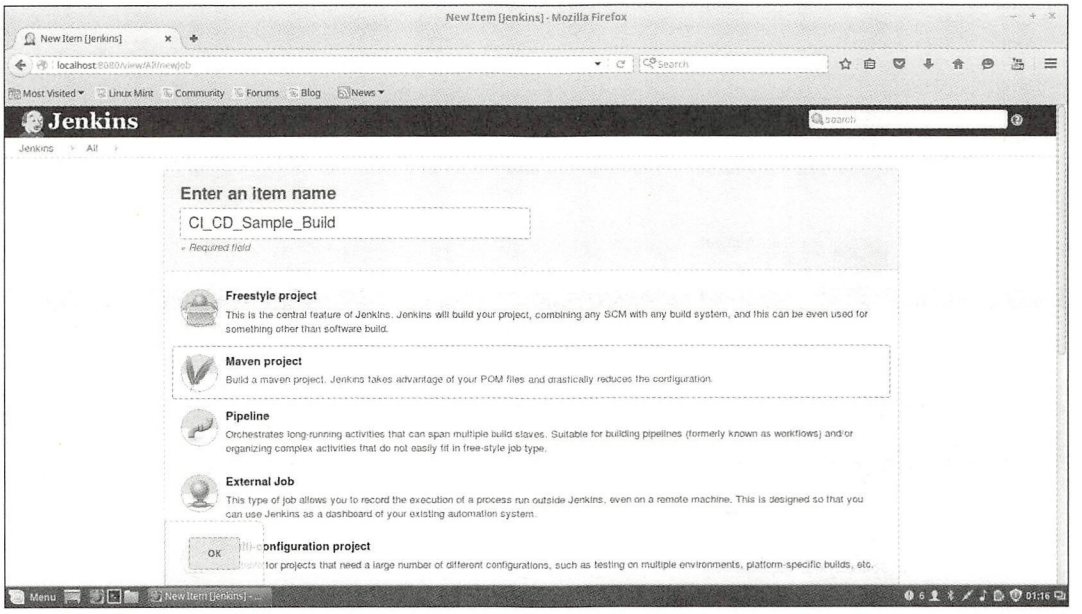


图 7-6

单击 **Create a new job**。选择 **Maven project** 并为其命名。单击 **OK** 按钮后，就可以在出现的作业配置窗口中进行相关配置了，如图 7-7 所示。例如，在 **Description** 框中加入一些描述，如“该任务负责构建通用组件模块”。

下一步是配置 Git 代码库的相关信息，告诉这个作业要从哪里提取需要构建的代码，如图 7-8 所示。对于 Maven 工程而言，Jenkins 将在工程的根目录下查找 pom.xml 文件。

在 **Repository URL** 框中，需要指定 GitHub 代码库的 URL。然后，我们可以在 **Build** 部分指定命令的参数，如发布平台等希望在模块中执行的部分，如图 7-9 所示。

图 7-9 中在 Goals and options 指定的参数如下所示：

```
-Dresume=false release:prepare release:perform -X -Darguments="-Dsource.skip -e -
Dmaven.javadoc.skip=true verify"
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

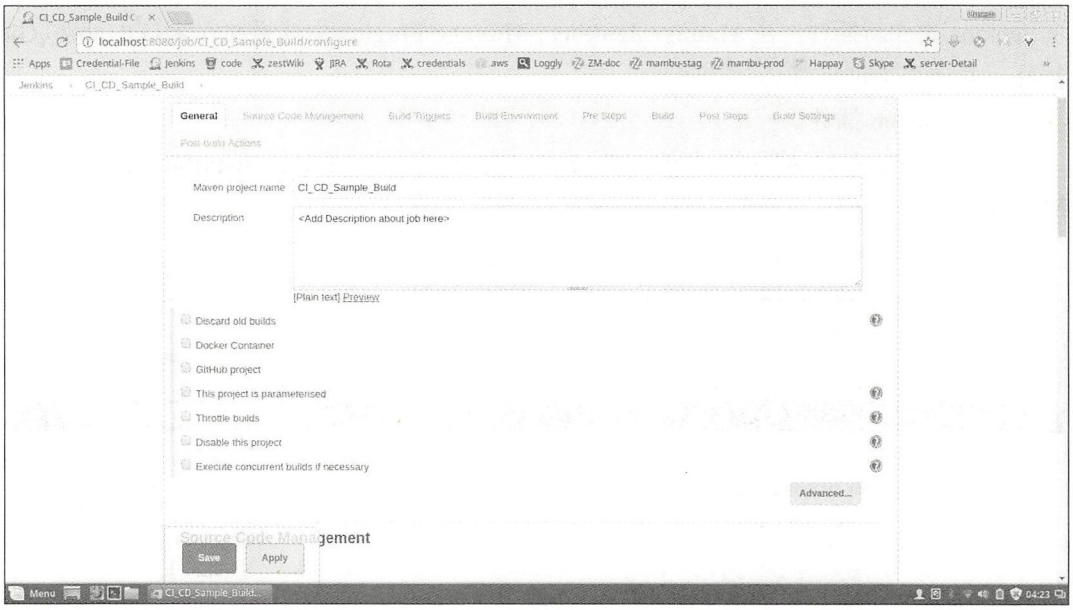


图 7-7

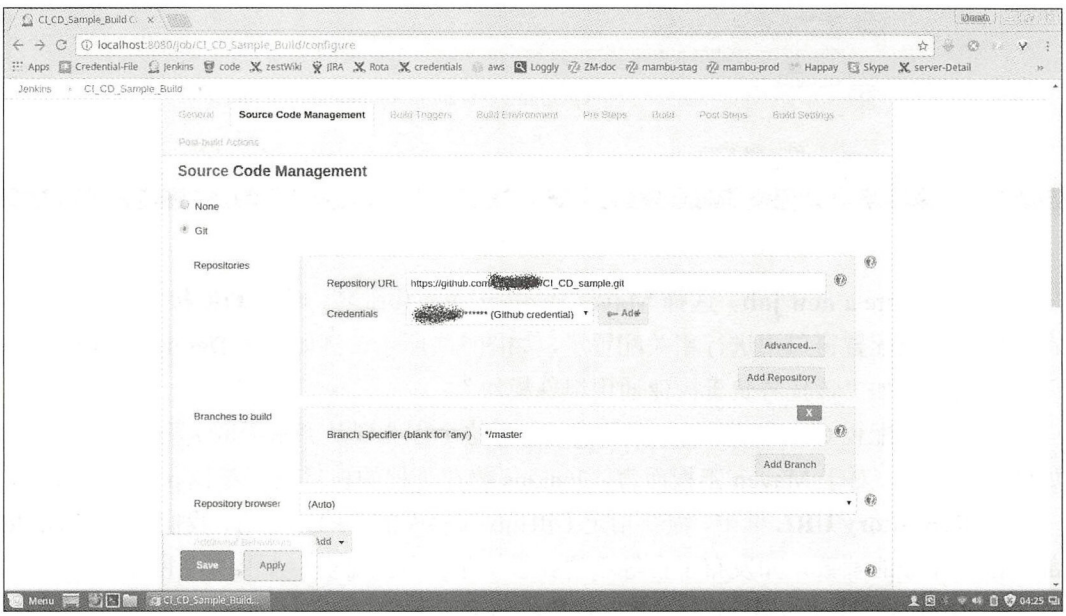


图 7-8

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

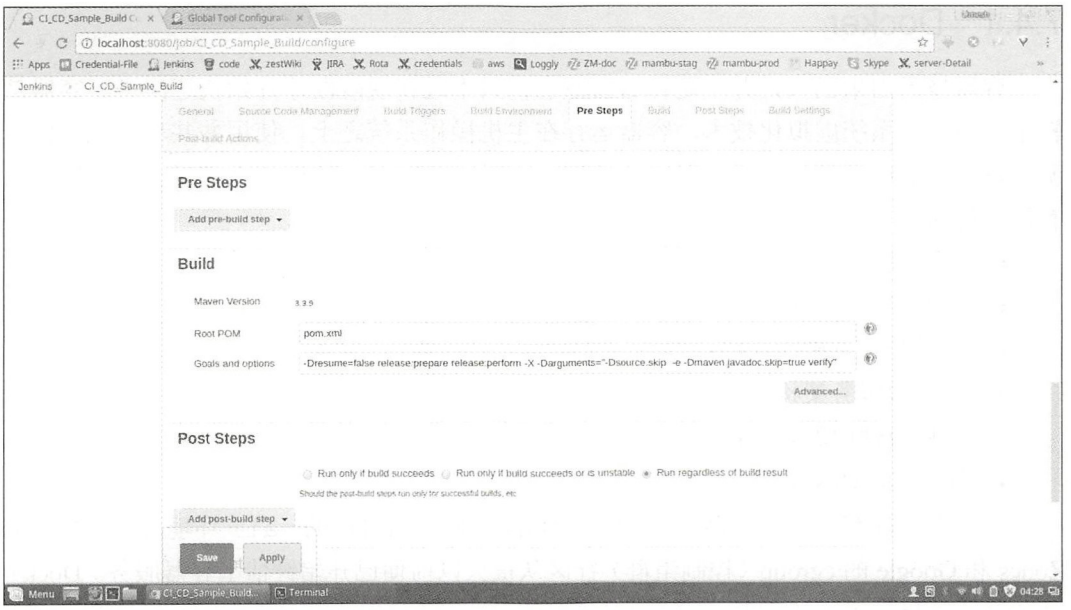


图 7-9

这段参数的意思是准备打包发布代码，并在这个构建版本中跳过文档生成环节。使用 Maven 来进行打包构建可以确保对应用开展一些检查，从而保证：

- POM 文件中不存在对任何 SNAPSHOT 版本的依赖；
- 代码中不存在未提交的改动；
- POM 文件的版本号已经由 x-SNAPSHOT 更新为 SNAPSHOT 版本；
- 使用版本名对 SCM 中的代码做标记（按提示输入）；
- 在 SCM 中提交新的 POM 文件。

上述步骤中任何一步失败，都将导致发布准备阶段失败，从而导致最终构建失败。

至此，我们可以准备运行第一个基于 Maven 构建的 Jenkins 作业了。

7.4 微服务的 Docker 化

Docker 日益受到欢迎。它能帮助开发人员创建规模更小的模块化组件，并且在 Docker 中进行打包和分发代码也很简单。这些也正是 Docker 同样适用于微服务平台的原因。我们接下来将会探讨如何在微服务中使用 Docker。但在此之前，让我们先来了解一下什么是 Docker 以及如何安装 Docker。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

7.4.1　Docker

容器技术由来已久，并不是什么新概念，只不过在现阶段才开始快速演进。容器是轻量级的操作系统虚拟化技术，容器运行在主机操作系统之上，使用并共享主机内核资源。每个容器均可独立运行，但必须通过一个层来共享基本的操作系统资源。容器的固有优势包括：

- 轻量级；
- 性能好，速度快；
- 交付基础设施即代码；
- 支持微服务架构；
- 可移植性问题较少；
- 简化 DevOps 工作。

有些企业拥有自己的容器版本。例如，BSD 一直使用自己的 `jail` 环境。此外，Solaris Zones 和 Google 的 cgroup（控制组群）社区从很久以前便已开始钟情于容器服务。Docker 是开源社区最常采用的容器技术，这使 Docker 成了容器化技术事实上的标准。作为一个容器，Docker 继承了上文提到的容器的所有优势。Docker 是一个轻量级层，运行在主机操作系统之上。Docker 最初使用的是 **Linux 容器组建**（Linux container components，LXC），但现在使用的是 **runC**。下面是 Docker 的一些主要组件。

7.4.2　Docker 引擎

Docker 引擎是位于主机操作系统之上的层，对 Docker 客户端和 Docker 守护进程而言可称为 **warper**。该引擎负责编排容器。开发人员或应用程序等最终用户与 Docker 客户端进行交互，然后 Docker 客户端将这些命令再传递给 Docker 守护进程，再由 Docker 守护进程去执行命令。

7.4.3　Docker 镜像

Docker 镜像是一个只读模板，用于创建和保存 Docker 镜像以供以后使用。Docker 文件定义了镜像中应该包含的内容。这些镜像用于创建 Docker 容器，是构建 Docker 容器的基本组件。

7.4.4　Docker 存储

Docker 存储使用多层文件系统 **AuFS**（高级多层统一文件系统），并负责管理网络。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

AuFS 基于**写时复制**（copy-on-write，CoW）的概念，同时带有一个只读层和一个只写层，二者在操作完成后再将结果合并。

7.4.5 应用程序在 Docker 中是如何工作的

通常情况下，开发人员会将一组命令写入 Docker 文件中，Docker 文件会指明所使用的基础镜像，例如，使用哪个版本的 Linux 系统，以及应用程序运行所需的 Web 服务器或其他工具。这些基础镜像可以在 Docker Hub 上很容易找到。系统在执行了 Docker 文件中的指令后便会建立 Docker 镜像，并将 Docker 文件中的先决条件（例如复制应用程序的可执行代码或配置等）注入 Docker 镜像中。接下来，生成的镜像根据需要可以交付到任何环境中。如果这是未来发布产品所使用的基础镜像，则可以将其保存在代码库中。

7.4.6 公共、私有和官方的镜像库

公共镜像库用于存储最常用的 Docker 镜像。它在相关社群推动下不断发展演进，对 Docker 的兴起帮助很大。镜像也可保存在私有镜像库中。开发人员可自由选择将自己使用的 Docker 镜像上传到公共镜像库或私有镜像库。官方镜像库由 Docker Hub 组织和推广，主要是操作系统库或语言运行库。

7.4.7 Docker 与 VM 的区别

虚拟机（virtual machine，VM）和 Docker 的工作环境相差迥异。Docker 相对 VM 而言属于轻量级应用。VM 和 Docker 的主要区别在于，VM 提供硬件级别的虚拟化，而 Docker 提供操作系统级别的虚拟化。下面，我们来进一步探讨 VM 的工作原理。图 7-10 描述了 VM 和 Docker 在内部层级方面的主要区别。

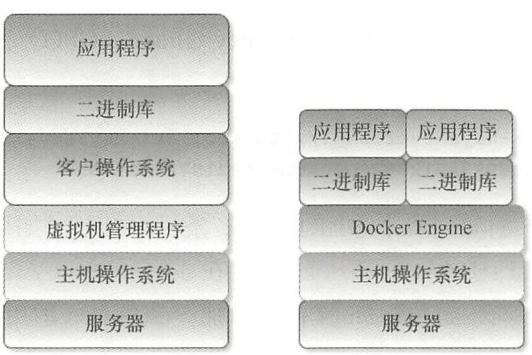


图 7-10

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

VM 是指使用虚拟机管理程序，在主机操作系统上运行的操作系统的镜像。每个 VM 本身，包括它使用的镜像和所需的应用，都运行在一个完善的操作系统中。由于 VM 不直接操作硬件，所以任何硬件请求，甚至包括通过键盘来获取用户输入的操作，都会通过正在运行的应用来调用客户操作系统，然后通过客户操作系统调用虚拟机管理程序，再通过虚拟机管理程序调用主机操作系统，最终抵达最下层的物理硬件。这样的调用流程会导致操作响应上的延迟。一个 VM 可能有数 GB 大小，从而导致启动时负载较大。如果同一台物理主机中的 VM 数量从 1 个增加到 3 个，就会显著减慢 VM 的运行速度。服务器上 VM 数量的增加将导致该主机所运行的操作系统数量增加。由于 VM 规定不能在不同 VM 间共享客户操作系统，因此每一个 VM 只能运行自己的客户操作系统，这会导致主机操作系统资源的浪费。

与 VM 的工作原理不同，Docker 可以在服务器中的所有容器之间共享主机系统内核资源。Docker 系统的抽象化是在用户空间层中，没有单独的内核负载，因此能够降低内存及 CPU 等硬件资源的消耗。启动整个 VM 一般需要几分钟时间，而启动 Docker 则只需几秒钟。Docker 的唯一限制因素是其无法像 VM 一样在容器内运行与宿主机完全不同的操作系统。简而言之，完全隔离只能由 VM 实现，而 Docker 可以实现快速高效构建共享内核的高性能的容器。

7.4.8　在 Linux 中安装 Docker

在 Linux 3.13 内核版本以上才可以运行 Docker。运行之前请先检查内核版本，如果低于 3.13 版本请及时升级。下面说明如何将 Docker 安装在 Linux 设备上。

（1）登录到设备控制台。确保拥有 sudo/root 权限。

（2）通过 `sudo apt-get update` 命令更新 `apt-package` 信息。

（3）通过 `sudo apt-get install apt-transport-https ca-certificates` 命令安装 CA 证书，以确保在 HTTPS 下可以工作。

（4）在 Linux 中添加一个新的 **GNU Privacy Guard**（GPG）密钥，用于实体间的安全通信。

```
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

（5）在 `apt-get` 的源列表中添加 Docker 条目。打开（如果不存在则创建）文件 `/etc/apt/sources.list.d/docker.list` 并在 `deb https://apt.dockerproject.org/repo ubuntu-xenial main` 文件中增加一个条目（适用于 Ubuntu 16.04）。操作完成后保存文件。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

（6）再次运行 sudo apt-get update 命令。

（7）建议安装 linux-image-extra-*，以使 AuFS 存储器正常工作。

（8）运行 sudo apt-get install docker-engine 命令在系统中安装 Docker 引擎。

（9）如果想以守护进程来启动 Docker 引擎，运行 sudo service docker start 命令。

现在 Docker 就已经安装好了，并且已经作为守护进程在后台运行。这可以通过运行 docker run hello-world 命令来测试一下，将触发 Docker 在本地查找 hello-world 镜像。如果在本地没有找到该镜像，Docker 将会前往 Docker Hub 提取镜像并将其保存在本地，然后再从本地运行它。

Docker 将基础架构视为代码，即由各种命令或脚本组成的 Docker 文件。在部署阶段，这些命令是按先后次序逐个执行的。就像代码中的头文件一样，Docker 文件是通过 FROM 命令来说明所引用的基本镜像以及从哪里构建镜像。该镜像可以是私有镜像，也可以是公共镜像。下面，我们来看看 Docker 文件如何能帮我们构建镜像。

GitHub 有许多关于如何使用 maven-docker 插件的选项，每人可以根据个人喜好不同来选择不同的插件来使用。本节以 Spotify 为例来说明创建 Docker 镜像的操作过程。

（1）创建 Maven 包。

（2）构建 Docker 镜像。

（3）将 Docker 镜像推送到 Docker Hub。

在这之前还需要在微服务工程中做两项额外的配置，即在 pom.xml 文件中指定 spotify 插件（及其各种属性），并在工程中指定 Docker 文件。spotify 插件可以帮助运行 Docker 文件来创建一个镜像，而 Docker 文件则告诉系统如何在 Docker 镜像中运行这个工程。

通过以下配置在 pom.xml 文件中添加 spotify 插件。

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <version>0.4.11</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                    <goal>build</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <dockerDirectory>${project.build.directory}/classes</dockerDirectory>
            <imageName>Demo/${project.artifactId}</imageName>
            <resources>
                <resource>
                    <targetPath>/</targetPath>
                    <directory>${project.basedir}</directory>
                    <excludes>
                        <exclude>target/**/*</exclude>
                        <exclude>pom.xml</exclude>
                        <exclude>*.iml</exclude>
                    </excludes>
                </resource>
                <rescource>
                    <targetPath>/</targetPath>
                    <directory>${project.build.directory}</directory>
                    <include>webgate.war</include>
                </rescource>
            </resources>
        </configuration>
    </plugin>
</plugins>
</pluginManagement>
```

请我们试着理解一下上面所说的插件。在这个插件中我们看到有几段代码。例如，在<execution>标签中，我们把<phase>的值设为 package，并把<goal>的值设为 build。下面我们来看 POM 文件中的另一个重要配置项：

```
<dockerDirectory>${project.build.directory}/classes</dockerDirectory>
```

这里我们指定了 spotify 如何在工程中查找 Docker 文件：

运行下面的命令来执行打包操作：

```
$ mvn package docker:build
```

这条命令首先运行 mvn 包，以构建镜像，但不会将生成的镜像推送至 Docker Hub。我们可以通过两种方法将 Docker 镜像推送到指定的代码库。第一种方法是在插件的配置部分插入如下代码行：

```
<pushImage>${push.image}</pushImage>
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第二种方法是在命令行中给出选项，构建和推送镜像的整个命令如下：

```
$ mvn package docker:build -Dpushimage
```

在默认情况下，如果使用以下属性，则会把镜像推送到指定组织 yourCompany 下的 Docker Hub：

```
<docker.image.prefix>yourCompany</docker.image.prefix>
```

必须将这个前缀添加<imageName>标签中：

```
<imageName>${docker.image.prefix}/${project.artifactId}</imageName>
```

yourCompany 的值在 Docker Hub 中应该被声明为组织机构，否则推送时就会失败。当然也可按照如下方式将镜像推送到私有的 Docker registry：

```
<docker.registry>docker-registry.alooma.com:5000</docker.registry>
```

可以使用以下命令来更改镜像名称：

```
${docker.registry}Demo/${project.artifactId}
```

也可以考虑将 Commit ID 附加至镜像名称中来为镜像命名。有很多插件（取决于构建工具）都能帮助获取最新的 Commit ID。配置如下：

```
<resources>
    <resource>
        <directory>${project.basedir}</directory>
        <filtering>true</filtering>
        <includes>
            <include>**/Dockerfile</include>
        </includes>
     </resource>
</resources>
```

上述<resource>标签的内容是让 Maven 在这个模块中包含创建镜像的 Docker 文件，Docker 文件应驻留在此，类似于下面这样：

```
FROM maven:3.3-jdk-8-onbuild
```

当然，如果想在 Docker 启动时在 Docker 中运行指定的命令，可以增加下面的代码行：

```
CMD ["java", "-jar", "/usr/app.jar"]
```

到现在为止，微服务的 Docker 化已经完成。接下来可以配置 CI 工具来搭建持续集成流水线。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

7.5　在 Docker 化的微服务中使用开源 CI 工具

微服务的目标始终是推进任何新功能或代码库的快速交付和部署。要想做到这一点，我们必须根据这个目标来配置 CI 和 CD 环境。此外，微服务 Docker 化还要求 CI 和 CD 环境能够适应快速交付和部署。在本节中，我们通过使用 Jenkins、Maven 和 Docker composer，来演示如何为容器化的微服务搭建持续交付流水线。基本操作过程是构建微服务的 Docker 镜像，将其推送到代码库，然后使用该镜像运行 Docker 容器，具体步骤如下。

（1）开发人员将代码推送到 GitHub，再由 GitHub 通知 Jenkins 作业。

（2）Jenkins 从 GitHub 中提取代码，包括开发人员提交的 Docker 文件。

（3）Jenkins 编译源代码，运行单元测试用例，然后在 `maven-docker` 的帮助下构建和打包应用。

（4）部署最近一次成功构建的软件包。

（5）使用 Docker composer 构建一个 Docker 镜像并对其进行测试。

（6）待上一步操作成功完成之后，将该镜像发布到 Docker 代码库。

将 Docker 集成到 CI 流水线中有很多好处，例如，可以提高构建效率、缩短作业时间、增加运行的作业量、消除语言技术栈的限制并能提高基础架构的整体利用率。Docker 容器在环境间的可移植性非常好，无论是开发人员的笔记本电脑还是生产环境的服务器，都可保证部署时的一致性。

待容器成功构建和配置之后，需要运行一系列的集成测试来确认服务是否正常运行。测试成功后，镜像将推送至 Docker Hub registry 中。请注意，我们在这里假设 Docker 文件将被提交到版本控制系统中。例如，使用 GitHub 作为**版本控制系统**（Version Control System，VCS），因为它提供了很多内建的功能。例如，通过 GitHub 的 Webhooks 功能来设置触发器，与 Jenkins 的任务进行集成。Webhooks 功能位于 GitHub 的 **Settings** 页面中的 **Webhooks and Services** 设置项下面。

在这个示例中，Jenkins 将使用主从架构。GitHub 插件需要安装在 Jenkins 的主节点上，它始终轮询指定的 GitHub 代码库。每当发现代码库有新的变更时，便会触发一个 Jenkins 作业。Jenkins 的从节点将对 Docker 容器中的镜像打包运行所有测试。Jenkins 的从节点应该具有 Java 运行时环境，将 SSHD 服务（SSH Daemon）激活，并且确保 Docker 引擎服务正常运行。

在 Jenkins 上面创建一个作业并只允许该作业运行在新创建的 Jenkins 从结点上。该

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

作业应使用 **Poll SCM** 选项，时刻监控 GitHub 是否有新的提交。Poll SCM 选项是与 GitHub 代码库中的 Webhooks 协同工作的。在设置完成后，就可以开始编写该作业的 shell 脚本了。脚本应该包含以下几行内容：

```
# build docker image
docker build --pull=true -t <Your_Repository_Path>:$GIT_COMMIT

# test docker image
docker run -i --rm <Your_Repository_Path>:$GIT_COMMIT ./script/test
# push docker image
docker push <Your_Repository_Path>:$GIT_COMMIT
```

让我们来对这 3 行命令进行逐条讲解。下面这行负责制作 Docker 镜像：

```
# build docker image
docker build --pull=true -t <Your_Repository_Path>:$GIT_COMMIT
```

- `docker`：调用 Docker 引擎。
- `build`：用于初始化镜像构建流程的命令。

具体选项说明如下。

- `-pull`：表示始终提取最新版本的镜像。
- `-t`：表示代码库的名称以及可选的镜像标签。

这里，镜像名称被指定为 Git 的提交 ID。`$GIT_COMMIT` 是 Jenkins 的环境变量，GitHub 插件在运行命令时会自动为其赋值，所以我们不用关心如何填充它。

第二行是指在 Docker 镜像中运行测试用例：

```
# test docker image
docker run -i --rm <Your_Repository_Path>:$GIT_COMMIT ./script/test
```

- `docker`：调用 Docker 引擎。
- `run`：这个命令会先创建一个容器，然后再通过给定的脚本来启动该容器。

具体选项说明如下。

- `-i`：交互式选项，可以是`-interactive=false/true`。
- `-rm`：通知引擎在退出时自动移除容器。
- `./script/test`：这个目录下包含测试脚本。因此，Docker 引擎将在构建期间运行该脚本。这个测试脚本可以编写测试、集成测试等。

第三行则简单明了地告诉 Docker 引擎将镜像推送到代码库中：

```
# push docker image
docker push <Your_Repository_Path>:$GIT_COMMIT
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

7.6　小结

本章讨论了持续集成和持续交付，了解了这些术语的含义，知道了它们对于产品和服务的快速交付至关重要，以及它们如何与微服务架构完美匹配。快速交付是实现快速发布的前提。大规模的发布总是困难重重而且危机四伏。而微服务可以将一个大的发布分解成多个独立的较小的发布，因此任何服务都可以在任意时间发布新版本或回滚。要实现频繁发布需要在最新的持续集成代码和持续交付的流水线上开展自动化测试。本章还讨论了目前流行的持续集成和持续交付工具，然后围绕着 Jenkins、Maven 和 Spring Boot 技术提供了一些示例。本章还介绍了 Docker 的概念，并讨论了微服务应该如何使用 Docker。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 8 章
演进现有系统

到目前为止，我们已经讨论了很多关于微服务的内容。简单来说，这是一种可将大型系统分解成多个小型子系统的方法，使得整个系统的工作方式更为高效，更容易构建和维护。尽管这听起来很不错，但把一个平台分解为微服务架构模式将是巨大的挑战。无论是从头开始构建应用，还是将其从单体架构迁移到微服务架构，都会耗费大量时间和精力，并需要具备专业知识进行相关研究。

例如，有一个成功的单体应用，已经正常运行有些年头了。那么，我们为什么要改变它？因为它能处理许多业务，而业务的开展离不开消费者。随着互联网用户数量的增加，如何通过在数字化世界中提升我们的知名度，来吸引越来越多的流量来增加销售额是非常重要的。良好的数字化营销可以增加网站的访问量，而随着访问量的不断增加，网站的响应速度迟早有一天会开始变慢。对于一个访问量很大的电子商务平台，大促销带来的访问量的爆炸式增长就很难应对。因此，无论哪种曾经是这个平台的最佳选择模式或技术，随着访问量的增加，都不一定是未来的最佳选择。为了适应请求数量的增加，我们必须做出一些改变。微服务架构也许是最佳选择，当然这也取决于每个平台的实际情况。

另一个关键因素是平台中不同功能之间的依赖关系。更改看似与其他功能无关的某项功能时，很可能会导致严重问题，从而不得不重新部署整个应用，这意味着整个应用必须停止运行。

本章将尝试应对这些挑战，并且探讨可以遵循的最佳实践方法。本章将会覆盖下列主题：

- 演进微服务架构的主要机制；
- 举例说明如何演进示例应用；
- 面临的挑战。

典型的单体应用一般由 UI 层、控制器层、服务层、持久化层和数据库组成，如图 8-1 所示。客户端请求首先由单体应用直接处理。待完成某些业务逻辑或数据库交互之后，

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

单体应用将结果交付给客户端。

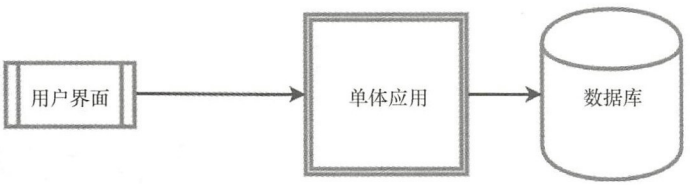


图 8-1

当平台流量临时性增加或永久增加后，可能需要向应用中添加多台服务器并在这些应用前面添加一个负载均衡器，如图 8-2 所示。

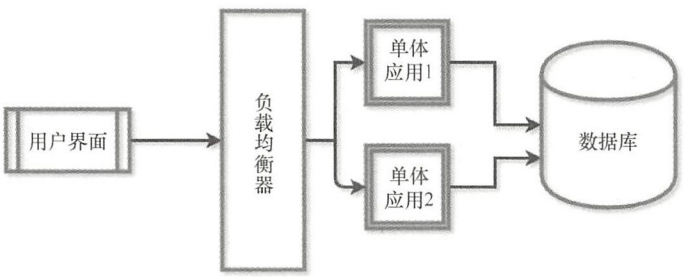


图 8-2

这其中的问题是，整个单体应用都需要进行扩展以应对增加的流量，但这些增加流量通常只作用于部分服务，并不会流向所有的模块或组件。一般来说，大流量往往只作用于某个子系统。但是，单体应用并不支持单独扩展某个子系统。要想扩展任何一个子系统，都必须扩展整个单体应用才能达到目的。微服务架构则允许只选择并且只扩展那些面向大流量的服务，而并非全部的服务，如图 8-3 所示。

实现图 8-3 所示的模式需要克服一些障碍。首先，我们需要聘请了解该领域的技术人员和领域专家，由他们去定义具有明确职责或界定上下文的不同子系统。

第二个障碍是团队的心态。团队对所开发的模块或服务需要拥有主人翁意识。

第三个障碍是 DevOps 团队会有很多开销并面临新挑战。在从单体架构迁移到微服务架构之前，DevOps 团队应准备好应对这些挑战。

第四个障碍与系统的监控、扩展和日志记录有关。无论服务数量是多是少，对其进行持续监控并在必要时及时采取应对措施都并非易事。我们将在下一章讨论如何解决系统的监控、扩展和日志记录问题。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

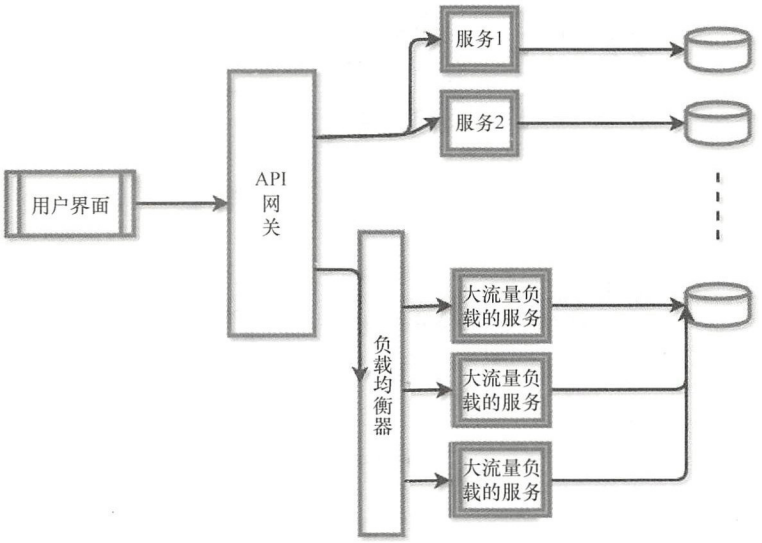


图 8-3

这是在演进系统时遇到的最严峻的几个挑战。除此之外，还会遇到其他诸多挑战，例如在第 6 章中讨论过的微服务测试，以及即将在下文中讨论的一些挑战。本章还会给出克服这些挑战所采用的解决方案和最佳实践。

8.1 从哪里开始

下面我们将分别从架构视角和数据库设计视角出发来逐一探讨微服务的演进方法。

8.1.1 架构视角和最佳实践

首先，架构视角的最佳实践是要找出性能最差或与用户交互最少的功能，最好能够找出不与用户直接交互而且用于执行离线作业的功能模块。因此在选择服务时必须十分谨慎。这个服务不应影响任何其他功能，并且代码与部署周期与其他服务完全独立。例如，这个服务可以使用与其他服务相同的数据库，但必须使用完全独立的部署流水线。

图 8-4 所示的服务层指的是许多功能的业务逻辑，例如某个调度作业服务或功能服务。因此，在确定某个服务可以作为完全独立的服务从单体架构中抽离之后，我们应该为其创建单独的部署流水线。这个服务可以与其他服务共享一个数据库，但必须可以单独运行，如图 8-5 所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

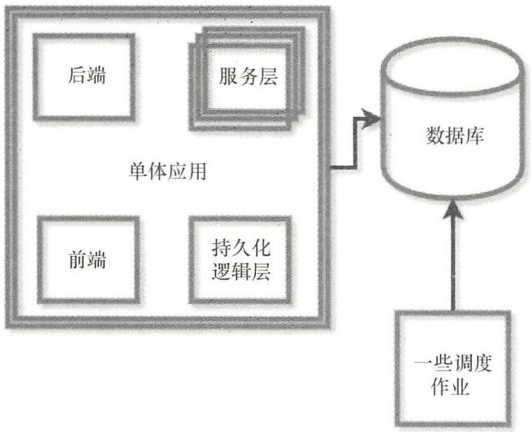


图 8-4

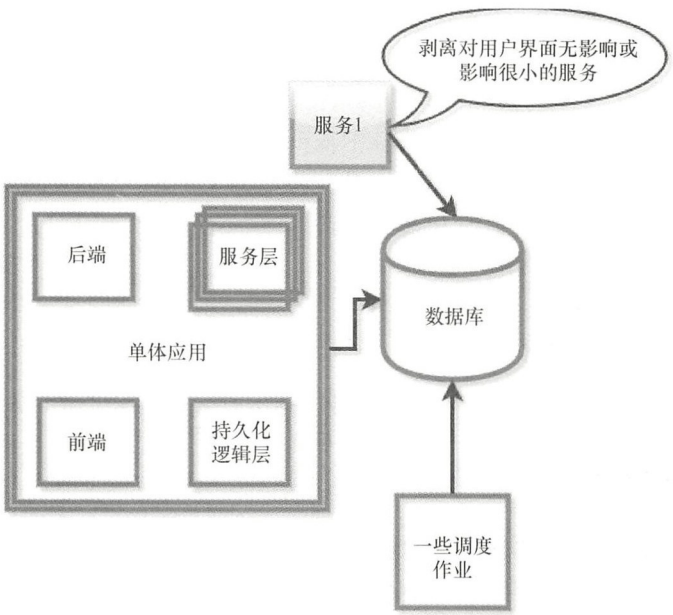


图 8-5

接下来在边界上下文中打破服务层的单体架构，让每项服务都拥有自己的数据库。这样一步步操作下去，就能在同一个平台中实现多个不同的微服务，如图 8-6 所示。

在此架构的演变过程中，基本准则是始终尽力抽离对单体架构影响最小的服务作为单独的微服务。当触及系统的核心或关键的微服务时，可以考虑在一段时间使用并行系统对外进行服务，即单体架构与微服务架构并存。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

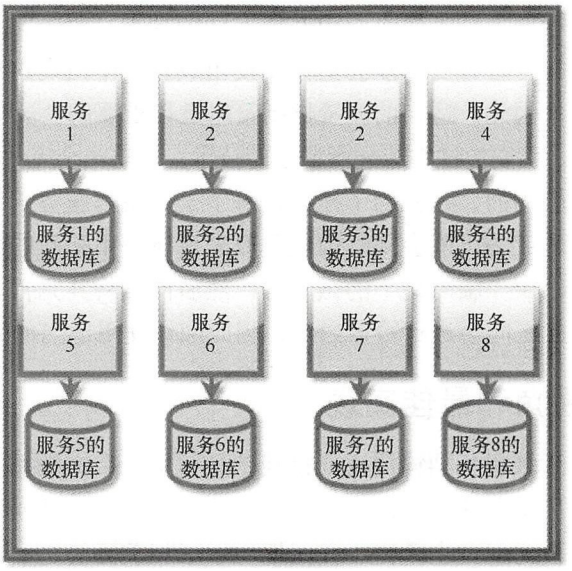


图 8-6

这种模式具有很多优点，还可以灵活地选择所使用的技术栈。但需要注意的是微服务的技术栈不要太过分散。如果在一个微服务架构中同时使用了多项技术，那么可能会引发一系列不同类型的问题。假设在用 Java 及 Python 等不同语言编写的多项服务中，都有类似的功能或某些类型的确认机制。那么某些有效性逻辑如需要改动，就需要将所有语言的有效性逻辑都更改完毕后才能正式进行部署。这将大大降低开发速度。

另一个需要注意的地方就是，切勿盲目使用陌生的新技术。面对众多的技术选项，开发人员有时会犯这种错误，例如，对于预估未来的目前并不存在的流量问题或技术问题而使用新技术。一般来说在系统的架构设计中，一般都会预留一些调整空间，以应对未来可能会出现的一些问题。但是，如果开发人员想得太多而盲目地添加新工具或新技术，很可能导致整个项目的失败。例如，开发人员使用一些新的缓存工具取代了传统缓存逻辑。如果新的工具配置不当，就会大大增加数据库的负载而并非减轻它的负载。因此，刚起步时不要使用太多的新奇模式，以免因环境过于复杂而引起死机。

无论何时，一旦决定从单体应用架构迁移到微服务架构，就应该对关键指标进行严密监控，如每秒请求次数、错误率和请求服务时间等应用程序级别的软件指标，以及 CPU 使用率、内存使用率和网络流量等服务器级别的硬件指标。在整个迁移过程中，应该始终对这些指标保持严密监控，直到微服务已经全部替换了单体应用为止。

除上述细节外，对于 API 请求还应该注意缓存策略和流量控制。在单体应用中一般会在适当的位置部署防火墙，因为内部应用调用多为函数调用；而在切换到微服务架构

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

后，可能会对不同的微服务或消费者应用开放不同的端点。由于某些未知原因或代码缺陷问题，某些客户端在一秒钟内可能会给系统发出成百上千次调用请求，最终导致系统阻塞，无法对外服务。因此，适当的流量控制策略是很重要的。

在微服务架构中，我们尝试实施分而治之的技术，即将一个庞大的、彼此之间密切联系的业务逻辑分割成较小的、独立的子模块，每个子模块都服务于该业务逻辑的一小部分。在迁移期间，部分业务可能会同时依赖于原来单体应用和新的微服务应用。这种情况下，一般需要创建一个“临时性”的微服务，然后慢慢将其完善为正式意义上的微服务。此外，微服务彼此之间会进行很多通信，因此应该使用适当的验证和授权机制。

8.1.2　数据库视角和最佳实践

从单体架构迁移到微服务架构的主要挑战是如何将一个大数据库分割成多个具有清晰边界的数据库。如上文所述，一般情况下需要领域专家提供帮助，因为只有领域专家才知道哪些数据通常会一起发生改变。在慢慢积累了相关经验之后，开发人员也可以具备这些领域知识，可以使用边界上下文来分割数据库。

第一步工作并不是真正开始分割数据库，而是使用数据库视图来确保做出合理的分割决策。对于新拆分出来的微服务，可以从数据库视图来读取或写入所需要的数据。虽然这期间可能会有些性能问题，但在真正部署这个微服务前都是可以接受的。另外一种方法就是把微服务所需要的数据从单体架构所使用的中央数据库迁移至微服务所使用的全新的数据库中。这种数据传输可以通过以下两种不同的方式来完成：

- 触发器（trigger）；
- 事件溯源（event sourcing）。

触发器可以部署在中央数据库中，那么当数据库及其各自表格中的数据被修改时，可以将这些数据发送到全新的微服务所使用的数据库，或者通过建立数据库链接来执行更新操作。另一种非传统方法是事件溯源，即利用某些工具记录当前数据库的事件，然后在新数据库中重放这些事件，达到数据同步的目的。Debezium 就是事件溯源的众多开源工具之一。

从单体架构向微服务架构的数据库迁移过程中，必须牢记向后兼容性与向前兼容性。任何的改动均不能破坏已经运行的功能。应该确保所有的东西都是设置好的，以便设有什么东西被关闭。例如，如果要从数据库中删除某一列，那么可以先将这一列重命名后发布上线，而不是直接删除。如果一切顺利的话，可以在下一次发布上线时将此列删除。微服务使用的数据库上所发生的任何迁移活动都不应该是手动操作，而应该要借助适当工具进行自动迁移，如 FlyBase 和 Liquibase。这两个工具都被多种语言所支持，并且也有 Maven 插件，可以与项目工程紧密集成。

从单体架构迁移至微服务架构并非一朝一夕之事，在完全迁移到微服务之前是一个

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

持续的过程。在这个过程完成之前，应该尽量避免在数据库中写入破坏性的语句，以防丢失数据。

8.2 示例应用及其演变过程

本节，我们将把一个示例应用演进为一个微服务应用，这个示例应用是一个销售服装、配饰及鞋帽等商品的在线电子商务网站。越来越多的消费者喜欢访问这个网站进行购物，所以必须在网站上以周甚至以日为单位来显示商品的价格信息。另外，这个网站每天都会更新一些新功能。

假设这个网站是一个大的单页应用（single-page application）。如果必须将该应用向微服务模式演进，那么首先应从服务的角度加以考虑。服务主要分为服务视图服务和后端服务两大类。第一步工作便是区分视图和后端（模型和控制器）部分。

视图服务是指用户能在页面上看到的内容，可进一步细分为静态内容和动态内容两部分。静态内容一般是 CSS 文件、常见的 JavaScript 文件等；而动态内容则是基于用户操作而生成的模板（由 HTML 组成）。例如，如果用户在界面上选择的分类是鞋，并且可以看到鞋子的 3D 图像，那么意味着系统提取了新模板并将其添加到了当前页面。这个新模板中除了模板本身外，还包括所需要的 JavaScript 文件和模板，模板中应该包含由静态内容服务器提供的 JavaScript 语句。视图服务允许模板服务根据需要提供更多项微服务，并且每项微服务都可以拥有自己的数据库。页面视图可细分为页头、促销信息、类别和目录以及页脚 4 部分，每部分的需求均可通过自身的微服务得到满足，该微服务取决于实现逻辑的复杂程度。

分离前后端代码带来的第一个好处是可以在不中断后端服务的情况下更改 UI 代码，例如，可以动态更改页头图片或优惠图片，还可以通过调度程序服务或定位发现服务让不同的城市可以显示不同的优惠信息和适当的图片，也可以根据时间来轮转优惠信息。当这个逻辑变得越来越复杂时，页面中的每一部分可以由不同的子团队负责管理。

对于后端服务而言，如果我们正在开发一个单体应用，可能会想到将所有代码集成在一起，构成一个大规模的混合体，以支持随时随地发生的大量的函数调用。但在微服务模式中，必须明确地界定微服务的上下文。对于这个网站，至少有以下 9 个微服务，具有不同的边界上下文和服务职责。

- 用户管理服务；
- 购物车/订单服务；
- 分类和目录服务；
- 优惠服务；
- 支付服务；

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- 发货/跟踪服务；
- 通信服务；
- 产品推荐服务；
- 调度服务。

所有上述服务均扮演不同的角色，同时每项服务都有自己的数据库。

8.2.1　用户管理服务

用户管理服务负责执行特定用户的操作，包括登录、退出、处理用户的详细信息、用户地址以及支付偏好等。登录或退出可作为一个单独的微服务（如验证服务）。不过这里讨论的是一个假设的应用，所以把登录和退出放在了一个微服务中。读者在处理自己的微服务时可自行决定是否将一个微服务变成两个微服务。但是，作者建议除非遇到延迟或性能问题，否则不要再次对服务进行细分。

8.2.2　购物车/订单服务

购物车/订单服务是用于维护购物车操作以及处理购物车中商品的删除或添加的服务，就像分类和目录服务是用来显示可用类别以及这些类别下的不同产品及其目录一样。优惠服务用于存储相关产品的优惠信息，例如，哪些优惠目前有效，以及适用于哪些地区等。优惠服务也可带有验证服务，例如，验证一辆购物车是否同时使用了两张优惠券，验证用户申请的优惠是否仍然有效或者是否适用于某类商品，等等。这些优惠及其有效性可能存在多种可能的组合，因此，开发人员应考虑将验证服务从优惠服务中抽取出来，作为单独的服务独立运行。但目前我们的示例没有必要这样做。

8.2.3　支付服务

顾名思义，支付服务是负责支付的服务。这个服务允许用户通过银行转账、信用卡、现金或 PayPal 等多种不同的方式来支付购物款。如果做得更好一些，可以实现一个包装服务（wrapper service），用来检查当前一体化的支付网关的状态。如果当前的支付网关不可用，则应该及时切换到备用方案来进行服务。支付操作是示例网站的业务关键点，千万不能在这个关键环节丢失用户，所以应该提供尽可能多的备用方案。

8.2.4　配送/跟踪服务和通信服务

支付服务完成后，另外两项微服务开始工作。一项是通信服务，用来向用户发送电子邮件或短信来确认订单，一旦订单被确认，还会给用户再发一封电子邮件，包括订单

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的细节和预计的送达时间。另一项是配送服务，用来启动发货流程并跟踪物流运输情况。

8.2.5 产品推荐服务

待用户完成购物之后，用户可以被重定向到另一个页面，通过产品推荐服务为其推荐与所购买的产品相似或配套的产品。

8.2.6 调度服务

调度服务用于执行后端任务，例如验证是否已经收到付款以及读取数据或转换数据来生成不同的报告等。

上述的每个微服务都拥有自己的数据库，并可以调用其他微服务，这让整个应用看起来怪怪的。如果完全从头开始构建微服务应用就很容易理解这种架构了。对于将应用从单体架构迁移到微服务架构这个过程，需要首先考虑将调度服务和通信服务抽离出来，作为单独的微服务独立运行。待这项任务成功完成后，再考虑分离分类和目录服务，以此类推。这个电子商务网站的微服务应用的最终架构图如图 8-7 所示。

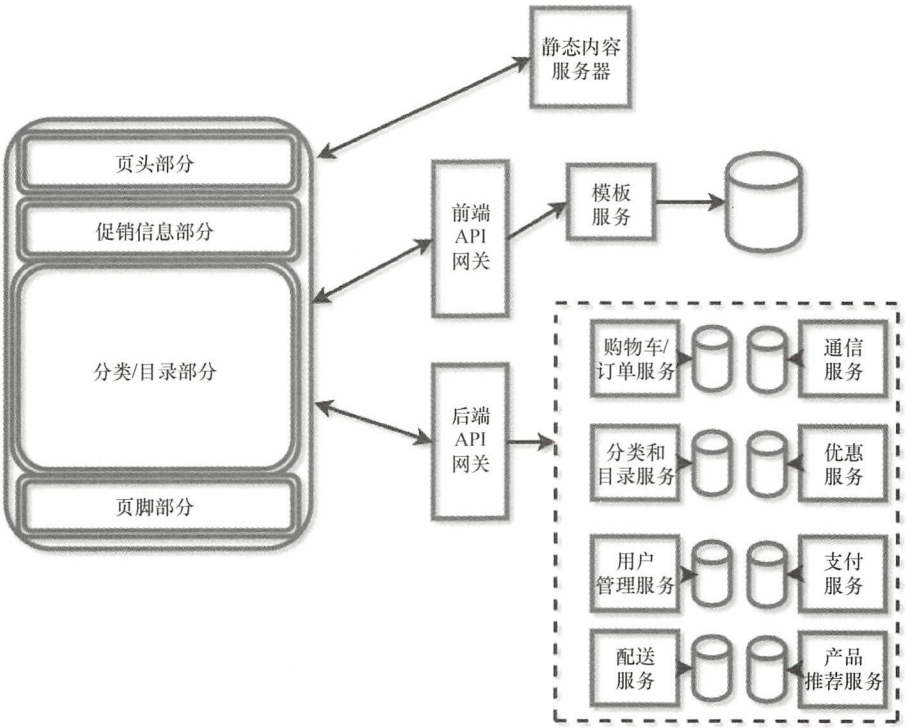


图 8-7

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

8.3　小结

从单体架构迁移到微服务架构是一项艰巨的任务，始终都需要一小步一小步做起，抽离不与用户直接交互的服务；密切关注应用程序级别和服务器级别的监控指标；尽量避免对数据库使用具有破坏性的语句；在数据库中使用新列并将旧列引用为失效列。领域专家必须帮助我们将应用分割成具有边界上下文的多项不同微服务。如果此类分割不够妥善或规划不当，那么拆分出来的微服务将会变得非常复杂。下一章将讨论微服务中另外一项重要工作——微服务的监控和扩展。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 9 章
微服务的监控和扩展

从单体应用迁移到微服务架构能给企业带来很多好处，例如，可以减小代码库，实施快速交付，并快速添加新功能，易于开发而且应用的扩展性更好，等等。但是，随着微服务架构的应用不断增加，其复杂性也随之增加，给企业实时洞悉系统的内部状况带来了异常严峻的挑战。系统中大量消息的流动使得外界很难知道系统内部的确切状态。此外，动态架构和自动扩展使情况变得更加复杂。本章将介绍一些与微服务监控和扩展相关的重要事项。

本章将会覆盖下列主题：

- 微服务系统的监控原则；
- 微服务系统的扩展原则；
- 微服务监控和扩展实践。

9.1 微服务系统的监控原则

监控是微服务控制系统的关键考虑因素之一。迁移到微服务架构，即从一个单体应用转变为复杂的分布式的微服务系统，在提供较大灵活性的同时也带来了不少挑战。微服务故障的概率和微服务或微服务节点的数量是成正比。由于不同微服务间的通信或交互流程都很难可视化，因此微服务的监控与微服务的开发同等重要。任何应用，无论是 Web 应用还是移动应用，都可能只向系统发出一个 API 调用请求；但在系统层面，这样的单个 API 调用可能会变成无数次内部通信。如果内部通信出现问题，哪怕只是一次通信出现问题，DevOps 和开发人员也必须追根溯源。由此可见，内部系统的可视化对于开发人员和 DevOps 采取快速补救措施是至关重要的。无论面对任何故障，团队都没有时间和精力从数兆字节的日志记录中手工查找问题的根源。因此，为了帮助团队迅速精准地找到问题所在，使用适合的工具来监控大量的微服务是很有必要的。

接下来，我们将讨论微服务系统监控领域的一些最佳实践。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

9.1.1　如何设置并使用警报

微服务架构中的各个微服务之间都是松耦合的。每个微服务的开发团队都对自己负责的微服务有着端到端的控制权。所以，微服务的警报的种类和阈值应由开发团队进行设计，而不是由 DevOps 团队主导，否则开发团队可能无法准确判断服务内部究竟发生了什么。因此，所有的微服务团队都需要自行决定并设计他们的警报、指标和仪表板（Dashboard）。同时需要注意的是 DevOps 团队应该能够查看系统的整体情况。

9.1.2　从一开始做好监控和发布渠道规划

监控工作不应发出无用的信息。有时候，开发团队会在欠考虑的情况下盲目设计出太多的警报，有些很重要，而其他的纯属多余。如此一来，开发人员便开始忽略多余的无用警报，同时也会忽略一些重要的警报。因此，监控问题应该是从项目一开始就需要认真考虑的，确保只创建重要的警报，并且以合理的投入来规划每项服务的警报。根据上述观点，开发人员应参与到警报创建工作中。另外，发布渠道也很重要。电子邮件等传统的沟通渠道太过陈旧，频繁发送的电子邮件基本上都会被自动过滤掉。所以可以考虑选择 Slack 等其他更流行的消息发布渠道。这类渠道不仅能够传递信息，而且还能给微服务的所有者分配任务。同时还应该根据团队的反馈，不断改进警报，停止发送无用的警报并逐渐添加重要的警报。

9.1.3　自动扩展和自动发现

这是一个自动扩展和自动服务发现的时代，服务按需扩展，始终处于动态变化中。因此，监控平台应该能够显示当前正在运行某项服务的服务器数量以及这些服务器的所有预定义的监控指标，如 CPU 负载、内存负载和流量负载等。这类指标还应该支持动态地添加和删除。此外，监控平台还应存储某类历史指标，以帮助了解流量的高峰期、运行服务的设备数量以及这些服务的利用率等。这可以帮助开发团队优化代码或自动调整运行参数。然而，这种监控系统在大规模部署时，可能会启动数百个代理服务，消耗了有限的系统资源。如果在微服务中直接嵌入监控代码就可以为运营团队省去很多麻烦。

在对微服务实施容器化处理时，系统架构的复杂性、整体的运营开销和监控负载都将随之增加。容器化技术凭借诸多潜在优势而成为微服务的首要选择。在扩展性方面，编排服务会根据实际需要不断地添加新容器，这给 DevOps 团队手动定义将哪项服务置于哪个容器之中带来了更加严峻的挑战。因此，要想在动态的容器原生环境中运行微服务，监控系统应该有能力去监控部署到任何地方的微服务，如 AWS CloudWatch 和我们

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的私有数据中心。

9.1.4 监控前门流量

我们应该小心地监控前门流量并经充分考虑后谨慎定义监控指标。前门的存在意味着微服务的入口点（entry point）暴露给外界，如一个 HTTP API，也可以是套接字或者静态网页等。在未加保护的情况下部署公共的或者可由合作伙伴访问的微服务会带来灾难性的后果——试想一下如果这个微服务受到了 DDOS 攻击的话会是什么样子。因此监控发布的微服务并且限制外界对它的访问就非常重要了。

监控系统能帮助了解系统外部发生的事情，例如，系统中活跃用户并不多的情况下，API 调用呈现突发性的增加和减少。如果监控系统能够通过所收集到的指标而自行分析就最好了。但一个监控系统最重要的功能之一还应该是显示有关服务性能的数据和数据流。开发人员需要借助性能指标来改进新版本的代码。性能指标包括但不限于 CPU 使用率、内存使用率、系统与数据库或其他服务的请求时间、请求处理周期以及响应时间与请求数量的对比图等。

9.2 监控模式的发展变化

监控是指必须由人读取的一些图形、警报或统计数据。这些警报可通过电子邮件或短信接收，也可在浏览器中进行查看。分析完这些警报，可能会采取某项措施进行干预。因此，警报可以自动发布，但要采取措施进行干预则需要人工介入，就像 Nagios 工具可能会在发现异常时自动发送警报一样。然而这种模式并不适用于微服务环境。随着系统中的微服务数量不断增加，我们需要实施智能监控。当系统发现故障或失效时，监控系统应该表现得足够智能来实施一些补救措施。如果监控系统发现本应正常运行的服务停止了工作，将会尝试重启这个服务。如果重启一再失败，那么，它将会分析日志并且可能会发现连接时间超时等一些常见问题，然后基于日志分析结果来执行既定操作。

图 9-1 展示了分布式微服务架构的通信情况。从图 9-1 中可以看出，各个微服务之间的通信非常频繁。要想持续监控诸如每秒请求数量、可用内存、活跃的连接数、无效的认证或过期令牌以及在哪个页面上丢失了多少用户等指标，都是极为困难的。但这些参数对于了解高峰期和低谷期的用户行为和应用行为是至关重要的。监控系统能将故障及时通知给运营团队。在上述的分布式环境中，查找问题的确切原因并给予纠正将需要很长时间，使得响应式监控（reactive monitoring）应运而生，在分布式的微服务环境下没有响应式监控是相当困难的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

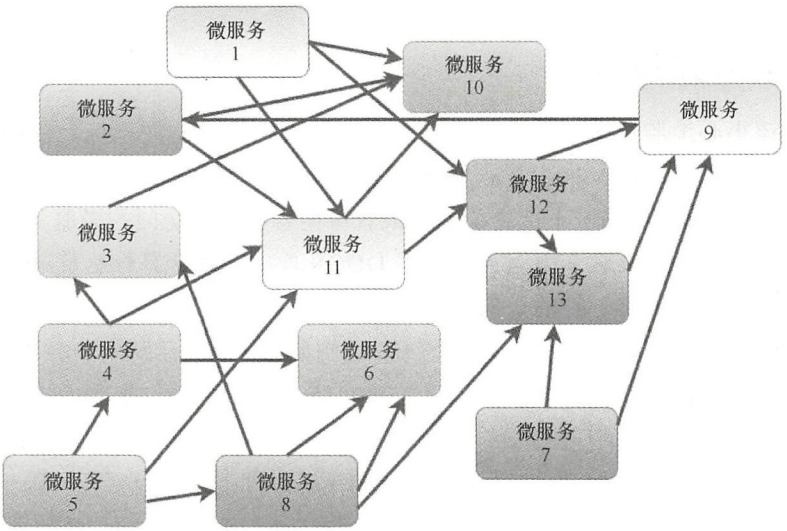


图 9-1

传统上讲，所有的监控工具通常都能满足 DevOps 团队需求。但这些工具的 UI 体验一般都不太好，有的工具甚至只能通过配置文件来进行相关设置。然而在微服务架构中，这些工具不一定专门由 DevOps 统一管理，而可以考虑进行去中心化处理，并分发给多个开发团队进行设置和维护。现代化的 UI 技术，让开发团队中的开发人员、质量保证人员和产品经理等不熟悉相关工具的非运维人员也可以快速上手，在遇到相关问题时可以给 DevOps 团队提供全面的信息。

快速交付是微服务的另一大优势，可以快速添加或删除任何功能。启用某项新功能时，可以采用并行发布，并且使用 A/B 测试的方法来快速获得用户对新功能的反馈。然后，产品研发人员将会基于测试结果来努力改进产品。这时，监控系统不是为了找出系统中的故障，而是用来观察并记录用户对新功能的反应，从而帮助产品研发人员更好地了解新功能产生的效果。这也会提高微服务监控的门槛。

对大数据、数据科学和微服务而言，监控微服务的运行时统计信息不仅能帮助了解应用平台的工作状态，还能帮助了解最终用户的偏好，以及他们对某些状况的反应。微服务的监控早已不是仅仅发送警报那样简单了。

9.3　日志记录有助于监控

当从单体应用迁移到微服务架构时，有时还会遇到堆栈信息不足的问题。例如，在

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

处理一个请求的过程中，服务 A 中发生了一个操作，服务 B 中发生了另一个操作，以此类推。最终这个请求处理以失败告终，但这期间究竟发生了什么事情，是哪个服务失败导致的，为什么还会有一大堆不知道的问题，为什么还需要开展大量的调试工作？如果某项微服务出错后就会抛出堆栈信息的话，问题定位会不会容易一些？一般来说为了方便调试，可以为每个请求都绑定唯一的全局请求 ID，来传递给每个服务并记录相关的日志信息。

绑定请求 ID 确实能帮我们了解不同的服务在处理特定请求时都采取了哪些动作，但日志可能位于不同的设备上，导致开发人员仍然难以追踪到问题根源。所以可能需要借助 Loggly 或 ELK 等工具对日志进行集中化处理。这部分将在工具部分进行更深入的讨论。

9.4 微服务系统的扩展原则

可扩展性是组织设计微服务架构时需要考量的重要因素之一。它赋予了开发人员足够的自由度来思考代码如何进行扩展以及具体的扩展策略。与扩展整个单体应用相比，在微服务中扩展任何指定服务可以通过更少的资源来完成。借助于微服务的帮助，企业可以快速灵活地扩展所需要的服务，进而能够从容应对平台中的负载变化。

Michael T. Fisher 和 Martin L. Abbott 在所著的 *The Art of Scalability* 一书中描述了真正实用的三维可伸缩模型——扩展立方体（scalecube）。

接下来为了更深入地了解可扩展性，我们将对图 9-2 中的每个坐标轴进行说明。

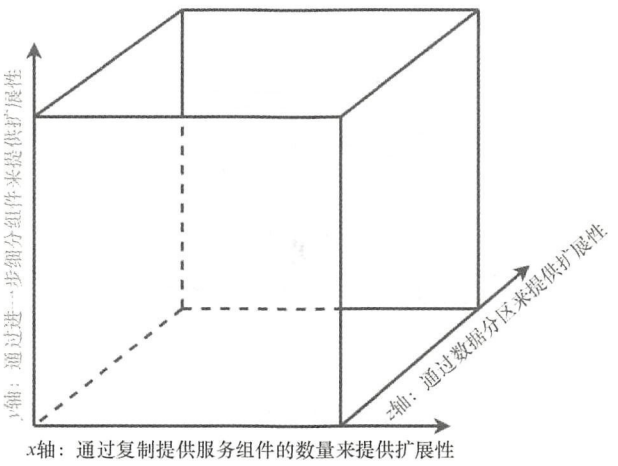


图 9-2

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

9.4.1　x 轴

x 轴是指在负载均衡器之后添加更多的机器数量以支持水平扩展。这种情况下只需要将微服务的代码复制到每台机器上来应对更大的流量即可。此类扩展性通常由 TCP 负载均衡器实现，如图 9-3 所示。这种方法的问题在于应用本身需要维护系统内部的状态列表，需要进行大量的状态处理和缓存工作。性能的好坏在很大程度上取决于缓存及其算法的表现。

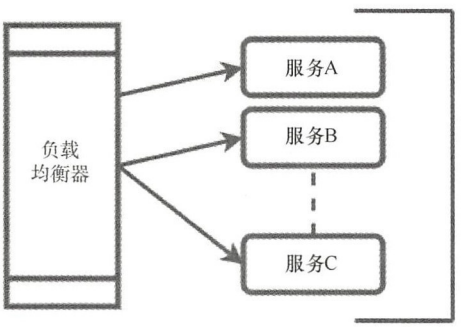


图 9-3

9.4.2　y 轴

若选择 y 轴的方法，我们需要将应用分解成多个规模更小的服务，如图 9-4 所示。这种方法适用于 REST 风格的微服务架构或 SOA 架构。

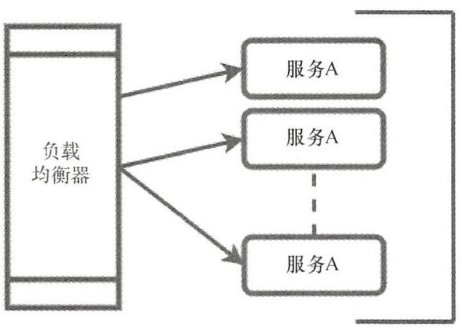


图 9-4

应用被分解成不同的小规模服务，而每个服务均需履行自己的一系列职责。基于登录和退出等动词进行分解是最常见的应用分解方法，另一种方法是基于客户、供应商和买方等名词来进行分解。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

x 轴的方法和 *y* 轴的方法也可以混合使用，如图 9-5 所示。

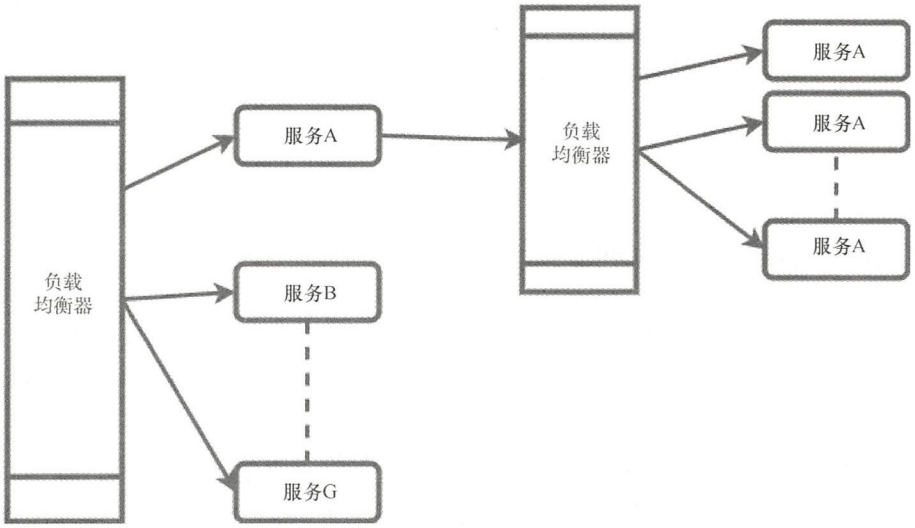


图 9-5

对于图 9-5 所示的这种选择，首先需要将服务分解成多个不同的子服务，然后再沿着 *x* 轴去扩展每项服务。第二个负载均衡器后面就是基于 *x* 轴的水平扩展。混合使用 *x* 轴和 *y* 轴的方法来进行微服务的扩展变得越来越流行了。

9.4.3 *z* 轴

z 轴的扩展方式在某种程度上等同于 *x* 轴的扩展方式，即每台服务器都会运行一个完全相同的代码副本。二者的主要区别在于 *z* 轴的扩展方式是基于不同的键值来向不同的服务器分发请求，如图 9-6 所示。因此，每台服务器都只处理部分数据，而不是全部数据。从根本上说，这种方法其实是 *x* 轴和 *y* 轴扩展策略的折中方案，即使用基于数据分片的方法。这种策略最常用于分片式数据库，但也可用于根据用户特征去扩展单体应用。

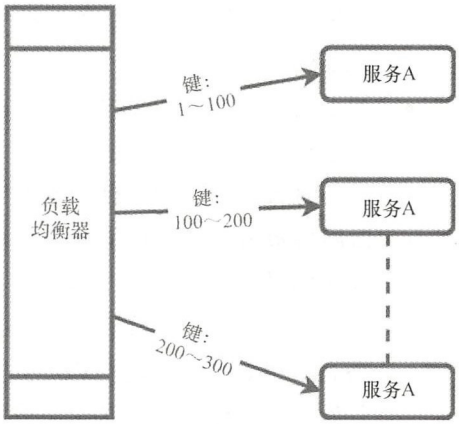


图 9-6

这种模式还适用于给某些用户或客户提供更高性能保障的高级访问策略，例如，视频服务网站可以保障高级客户的高速流媒体

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

传输不受网站其他因素的影响。这种方式的缺点是提高了应用代码的复杂程度。在这种模式下，路由也成了一个关键因素，因为路由算法需要随着数据量的逐渐增长而不断进行调整。

明智的扩展策略必须与应用架构相关联。y 轴的扩展方法与微服务架构最为匹配。然而为了提高微服务架构的扩展性能，必须在 y 轴后面复制或连接一个或多个其他轴所建议的方式综合使用。

9.5　实施扩展策略前请三思

仅仅创建微服务是不够的。软件工程的最终目标是要交付具有卓越性能的应用，打造流畅的用户体验，从而吸引更多的用户使用所提供的平台。用户数量的增加使微服务的扩展性变得重要起来。用户越多，微服务所需要的扩展比例就越大；高效的扩展性能够增强用户体验，从而吸引更多的用户来使用平台，继而推动整个系统再次进行向上扩展。微服务可以让一切都变得更加细化，包括按需扩展和按需管理峰值。需要注意的是，无论从客户还是最终用户的角度来审视微服务可扩展性的挑战，应用的自身性能始终都是至关重要的。

让我们再思考下在对微服务进行动态扩展之前需要考虑的一些关键因素。

- 基础架构是否支持动态扩展。如果基础架构支持动态扩展，那么可以继续进行操作；否则，请考虑基础架构是否支持手动扩展。如果仅支持手动扩展，则会影响扩展策略的一些决策。
- 扩展性所需遵循的策略。系统进行扩展需要遵循怎样的策略？这些策略可能取决于 CPU 使用情况以及网络流量等诸多因素。
- 如果在高峰期服务器扩展时有动态实例加入，则监控系统应该有能力去监控动态添加的服务实例。
- 基于微服务的架构是极度去中心化的，因为它专注于单个组件的成功复用，这有助于提高微服务架构的扩展性。在微服务架构中，由于每项服务均可独立于其他服务进行单独扩展，使得整个系统的扩展变得更加简单。
- 微服务数量的增加使得容器技术成为最有前途的微服务部署选项。目前市场上有许多的微服务容器化工具和设备可供挑选，像 Kubernetes 这样的开源容器工具便是不错的选择。主流云服务提供商也在其平台上增加对容器的支持，例如，Amazon ECS 和 Google Container Engine（GKE）。此外，这些提供商还纷纷出台各种机制，允许用户基于负载和业务规则来对基于容器的应用进行扩展。
- 容器化部署也会给扩展带来更大的挑战，如监控问题。大规模的容器监控可能会

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

导致严重的数据问题。由于需要监控的数据太多，给数据的收集、理解及对数据做出反应带来了极大的困难。

容器的扩展带来的挑战如下。

- 正如上文所述，容器数量可基于平台负载情况动态地增加或减少。因此，在这些容器销毁之前做好日志收集工作至关重要。日志收集现已成为容器进行扩展时需要考虑的重要因素之一。
- 另一个需要关注的问题是测试。大规模的容器化部署也会加剧容器部署本身的固有挑战。容器数量的增加使自动化测试、API 测试及通信性能等变得更加重要。
- 网络安全也是需要关注的问题之一，是系统扩展的另一个考虑因素。在不影响应用性能的情况下，如何规划或构建每个容器都具有一定级别的网络安全的方案也是很有挑战的。
- 最重要的一点是整个团队必须对使用生产环境的平台或容器有足够的了解。这里所说的团队不仅限于 DevOps 团队，而且还包括开发人员。开发人员也需要对容器技术有所了解，也应该知道容器内部的情况并进行调试。
- 微服务部署流水线的数量也在不断增加并且变得越来越复杂，大量的工作越来越依赖于 CI 和 CD 实现。在微服务的世界中，由于太多的不同服务陆续被单独开发出来，推动部署流水线数量出现显著增长。

调查显示，只有大约 19%的开发人员在生产环境中使用容器技术，其他团队则面临性能、测试和安全等诸多问题。云服务提供商正在借助容器服务进军微服务市场，试图通过容器化来解决所有问题。预计未来将有更多的团队会在生产环境中使用容器技术。

9.6 微服务的监控和扩展工具

监控工具不断地发展演变，目前有些先进的监控工具已经包含了响应式监控及用户行为监控等新特性。当系统发现故障时，应能够借助集中式日志记录等高效的工具来帮助调试定位问题。本节将探讨一些方便实用的微服务监控工具，包括 CloudWatch 等由云提供商所提供的一些著名的工具。

- **QBit** 是基于 Java 的微服务库，内含统计服务，可将数据注入 StatsD/CodeHale Metrics 之中。这项统计服务被称为统计引擎，可提供有关微服务的数据，从而帮助监控微服务。QBit 允许根据微服务统计引擎所提供的数据编写代码执行某项操作。StatsD 是用于汇总统计信息的网络守护程序，拥有面向 Java、Python、Ruby 及 Node 等语言的多个小型客户端程序库。
- **ELK** 是 Elasticsearch、Logstash 和 Kibana 的首字母缩写。Kibana 是可与大量聚

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

合日志数据进行交互的可视化界面。Logstash 是一款开源工具，用于收集、解析和存储日志以供将来使用。Elasticsearch 是集中式日志聚合和索引日志的服务器。Kibana 的工作原理是假设日志数据分析可以有效地发现问题。Kibana 支持 Fluentd 和 Flume 等多种日志聚合器。Kibana 最强大的部分则是将查询结果以图形和插件（widget）的形式直观显示出来。



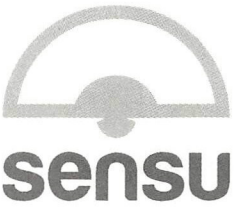
- **Dynatrace** 是应用程序性能管理工具，具有监控和用户体验分析功能，从而帮助开发人员、测试人员和运营人员确保其应用快速可靠地运行。Dynatrace 拥有 PurePath Technology 专利技术，能够端到端（浏览器端到数据库端）地跟踪每一个单独的事务。Dynatrace 用途广泛，例如自动依赖分析、自动建立性能基准、自动分析问题根因、自动发现容器化应用以及为云基础架构和 Docker 提供原生支持等。



- **Sensu** 用 Ruby 语言编写，是一个不到 1200 行代码的轻量级系统，可以执行异步检查任务并对系统进行更改。从本质上说，它是以消息传递导向架构 RabbitMQ 为基础的服务器/客户端架构（sensu-server 和 sense-client），凭借持续集成能力而拥有极佳的测试覆盖率，设计时考虑到兼容 Chef 或 Puppet 等现代化配置管理系统，专为云环境而生。Sensu 可通过开展一系列检查来监控任何事务。此外，它还能通过指定处理程序对问题/结果做出反应。Sensu 允许复用 Nagios、Icinga 及 Zabbix 等传统监控工具中的监控检查组件和插件并且提供一个 Web 仪表板，可以显示 Sensu 基础架构的当前概况并允许执行暂时关闭警报等操作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



- **AppDynamics** 于 2015 年 6 月推出了一款由机器学习所驱动的 APM 产品，用于监控、管理和分析如微服务等复杂架构。但这款产品的功能不仅限于 APM。从数据收集到处理，再从数据中获取知识，AppDynamics 可以全面准确地洞悉应用性能如何影响现有的业务。AppDynamics 可实时显示应用性能并自动发现应用拓扑和相互依赖关系，其 APM 工具还能提供分布式跟踪、拓扑可视化和动态标记功能。AppDynamics 通过 APM 工具和分析驱动的方法，提供大量的细节来帮助解决性能问题。AppDynamics 的代理程序非常智能，对于每一项事务，它们都知道何时需要捕捉重要的细节，何时只需简单地收集基本信息。



- **Instana** 成立于 2015 年 4 月，是面向网络级应用的监控和管理平台，拥有名为 Stan 的智能虚拟机器人助理。Instana 使用机器学习和数学算法来提供动态图形和图表，其智能机器人助理能向用户发送即时通知，以帮助用户监控和优化复杂的应用。



- **OpsClarity** 提供智能监控解决方案，可对网络级工具架构所特定的指标实施深度监控。OpsClarity 是用 Scala 编写而成，采用 Angular JS 实现 UI，Nodejs 作为服务器端。同时它还采用了机器学习库、图形库和大型流处理的基础架构。OpsClarity 支持的技术包括 Apache Kafka、Apache Storm 和 Apache Spark 等数据

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

处理框架，以及 Elasticsearch、Cassandra 和 MongoDB 等数据存储。



除上述工具外，Prometheus、Netsil、Nagios、New Relic APM、Foglight 和 Compuware APM 等也都是可供选择的监控工具。组织具体选择哪个监控工具完全取决于系统大小、组织需求和本身的财务状况。

- **扩展性**在微服务架构中并不那么简单，诸如 Netflix OSS 这样的框架和工具集会尽可能地自动处理新的微服务注册流程。如果再涉及 Dockers、Packer 及 Serf 等容器技术，事情将会变得相当复杂，但还是可以借助一些工具来扩展容器技术。下面是这些工具的清单。
- **Kubernetes** 是用于在主机群集之间自动部署、扩展和操作应用容器的开源平台，帮助构建以容器为中心的基础架构。Kubernetes 无须考虑每个应用在数据中心的哪个设备上运行，全部可自动完成。



- **Vamp** 是自托管式开源平台，用于管理依赖容器技术的面向（微）服务的架构。Vamp 负责路由更新、指标收集和服务发现等任务，可轻松编排复杂的部署模式。它与平台无关，是用 Go、Scala 和 React 语言编写的。



- **Mesos** 是具有动态资源分配功能的开源集群管理器，可用于管理工作负载。Mesos

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

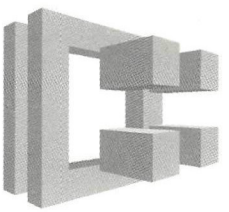
进程运行在所有的设备上，其中一台设备是负责管理集群的主设备，所有其他设备都是使用一定资源来执行某项任务的从设备。在 Mesos 中调度集群中的 CPU 和内存资源，就像 Linux 内核调度本地资源一样。



- **Consul** 是能为网络提供强大而且可靠的服务发现功能及许多其他功能的卓越解决方案，由服务发现、健康检查及键值存储的多数据中心等多个组件构成。Consul 是高可用的分布式系统，可在运行大量服务时帮助伸缩配置服务器。负责给 Consul 提供服务的每个节点均运行一个 Consul 代理，负责对节点上的服务及节点本身开展健康检查，确保其正常工作。



- **AWS 容器服务**，又称 ECS，是高可扩展的高性能容器管理服务，支持 Docker 等许多容器，可以轻松扩展自有的集群管理的基础架构。这项容器服务所涉及的主要组件包括 EC2 实例、Docker 守护进程和 ECS 代理等。ECS 负责管理 AWS 中的 Docker 容器，允许用户增加或减少 Docker 容器数量，同时可以评估和监控 CPU 使用情况。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

下面是可供选择使用的其他工具：

- etcd；
- Deis；
- Spotify Helios；
- Tutum；
- Project Atomic；
- Geard；
- Docker OpenStack；
- Coreos；
- Flynn；
- Tsuru（支持 Docker 集群）。

微服务的扩展工具种类繁多，其中有些支持容器技术，而有些不支持。诸如 Netflix OSS 这样的框架和工具集会尽可能地自动处理新服务注册流程。本节简单介绍了微服务各自领域中的各项工具。如想深入了解这些工具，可参阅大量的现成资料。

9.7　小结

监控和扩展是微服务架构中需要注意的两个重要因素，在微服务开始设计时就需要仔细考虑。本章还对响应式监控和扩展立方体进行了阐述。组织可借助很多工具来有效地实施微服务的监控与扩展，本章介绍了诸多工具。组织具体选择哪些工具，完全取决于平台大小、微服务数量、团队和用户数量或者组织的财务状况等。下一章将介绍在微服务环境中提出的一些常见问题以及如何解决这些问题的方法。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 10 章
故障排除

迁移到微服务架构可能有两种结果，一是迁移到微服务后的应用可能无法工作或无法按预期工作。解决办法是回滚到原来的单体应用，最终导致架构可能是介于 SOA 和单体架构之间的“四不像”。二是成功地从当前架构迁移到微服务架构。在第一种情况下，项目组将会再次被老生常谈的历史问题所困扰。在第二种情况下，历史遗留问题可能会得到基本解决，但会遇到一系列新问题。本章将探讨迁移到微服务架构后可能产生的新问题以及其解决方法。本章是第 9 章的延续，会利用第 9 章中介绍的工具来开展故障排除工作。

本章将会覆盖下列主题：

- 使用微服务时的常见问题；
- 解决或减轻常见问题的技巧。

10.1 使用微服务时的常见问题

微服务模式是将一个大的问题域分解成具有清晰定义边界的较小的组件，从而使得所面临的问题更加容易处理，并更有可能找到问题的解决方案。微服务架构的所有优势都会伴随着一些取舍，而这些取舍又会带来另一系列问题，例如，从微服务的扩展到微服务的监控，这些事情的难度都会成倍地增长。如果说监控单体架构已经很难，那么监控微服务则是监控单体架构难度的 10 倍，并且需要在更多的资金和规划上未雨绸缪。

本节将介绍采用微服务技术时通常会遇到的一些常见问题。

10.1.1 性能下降

性能下降是采用微服务模式所面临的最重要也是最常见的问题。在一个应用的完整的生命周期中，应用的性能是随时变化的。在微服务架构中，应用程序的代码分布在不

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

同的微服务组件中，因此应用的整体性能会在某种程度上受到影响。例如，某个用户请求可能会让我们的微服务组件执行 5 次不同的服务调用。由于任何一次服务调用都会产生一些延迟，因此会导致整个调用过程出现延迟。另一方面，如果其中某个微服务组件因为某种原因导致性能较差，并且该组件是多个其他组件的连接点，那么整个系统的整体运行速度将会明显减慢。

如图 10-1 所示，假设某个 Web 客户端发出请求，需要获取已登录用户的基本信息。假设基本信息的数据类似于 Facebook 上的个人资料，即包含帖子、回帖、图片及好友列表等丰富内容。为了通过一次调用获取这些信息，可以使用第 2 章中描述的 API 网关。第 2 章说明了在处理客户端发出的单个请求时，如何最大限度地减小客户端与服务器之间网络流量。服务器在 API 网关的帮助下将收集到的数据推送至客户端，而 API 网关则负责发出内部调用请求来获取所有数据，并将最终的响应发送至客户端。在获取这些数据的过程中，API 网关必须向 5 个不同的微服务组件发出 5 个不同的请求。如果有一个微服务组件由于硬件或其他原因导致过载或响应缓慢，就会导致系统整体性能的下降，从而使得客户端请求的等待时间延长。一行错误的代码就可能带来性能问题，并且很难识别出引发问题的组件。

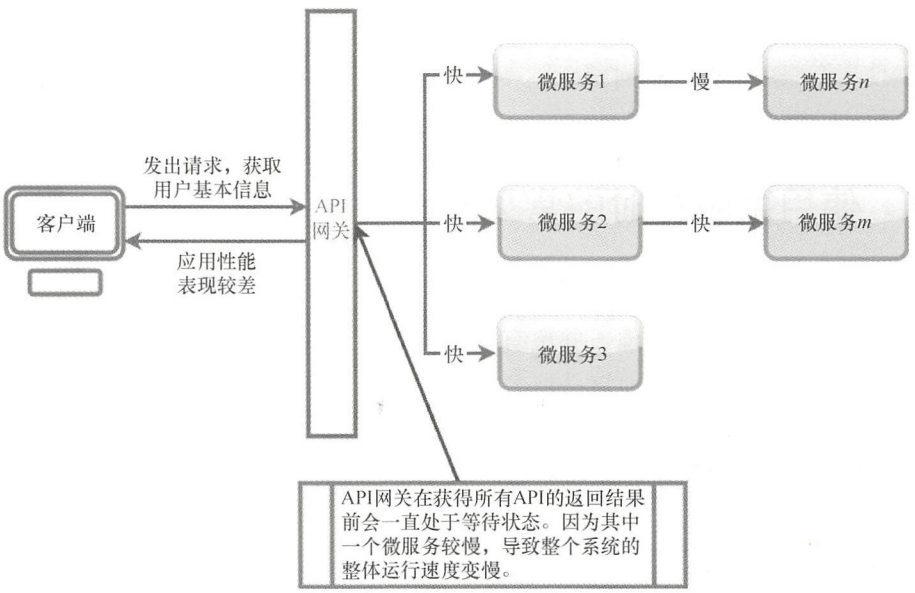


图 10-1

性能的好坏不仅指响应速度的快慢，还包括诸如可靠性和故障转移等多种因素，因此响应速度速度变慢并不是导致性能下降的唯一原因。可靠性和故障转移也是微服

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

务中常见的性能问题。例如，如果某个应用响应速度超级快却很不稳定，那么对于同样的一个合法请求，有时返回 HTTP 响应代码 200（代表成功），有时又会返回 404 错误，这也是一个性能问题。所以我们需要在速度和可靠性之间做出适当取舍。上述任何情况都会影响用户体验。应用的性能和可靠性二者很难兼得，需要进行取舍来保持完美的平衡。

10.1.2 日志记录位置因编程语言而异

一般来说，不同的微服务都运行在不同的服务器中。这些服务彼此隔离，所以它们的日志也放在不同的地方，并且很可能日志的格式大不相同。在调试某个调用时，开发人员都必须查看多个不同的日志文件及其他相关数据。就像图 10-2 所示的那样，微服务 1、微服务 2、微服务 3 直到微服务 *n* 甚至微服务 *m*，都运行在不同的服务器中，而且日志文件格式各不相同。对开发者而言，找出运行速度缓慢的那个组件并不是很容易的事。

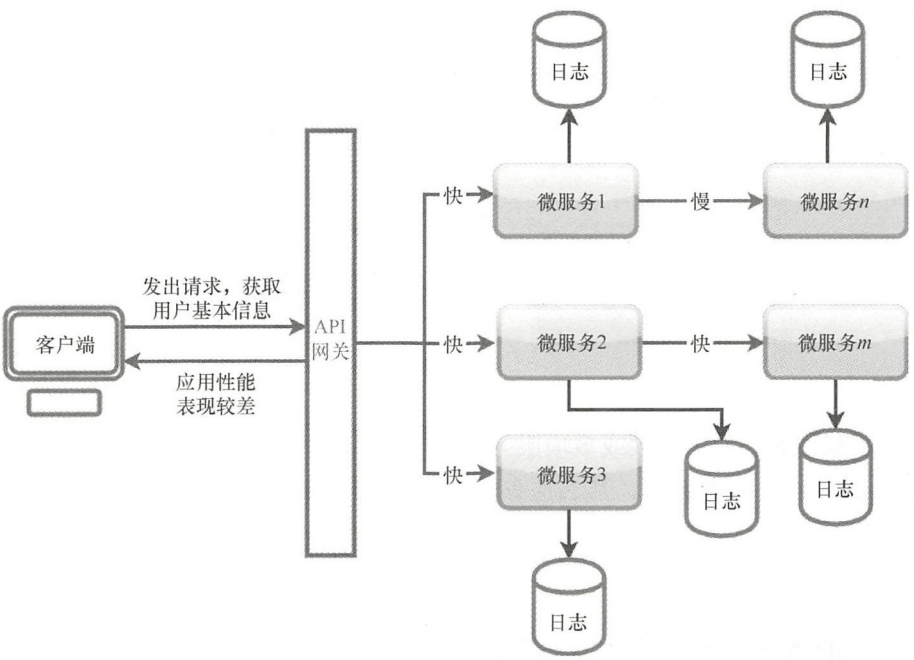


图 10-2

如果这个例子变得更复杂一些，例如，将一次调用所涉及的微服务数量从 5 个增加至 10 个，那么网站流量也会随之增多，并且根据日志记录等级的不同，日志文件的大小

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

可能也将增加到 GB 级。这让问题的识别变得更加困难。

10.1.3　多组件之间的耦合或依赖问题

服务的耦合或依赖意味着服务代码或库代码之间相互影响。然而根据微服务架构的原则，每个服务都应该是独立的，而且由于有明确定义的边界，每个服务不会对其他服务产生影响。尽管如此，对于通用共享的代码集，开发人员一般会开发一类共享库代码，然后将其作为辅助类引用到不同的微服务中。例如，某个组件为 API 的返回值而对某些数据实施序列化处理，之后这些数据在客户端作为同一个对象被反序列化操作。当我们需要对 4 个不同的组件应用相同的模式时，便很可能需要这种实践，即开发人员会制作一些共享库代码并将其用于所有组件。这会导致两个问题：第一种情况是如果通过一个共享库来处理多种常见操作，那么用于开发新服务的语言很有可能就是用于编写共享库代码的语言，从而限制了选择其他语言来编写其他微服务。这可能会导致所有的微服务均被限制在相同的编程语言上，如图 10-3 所示。

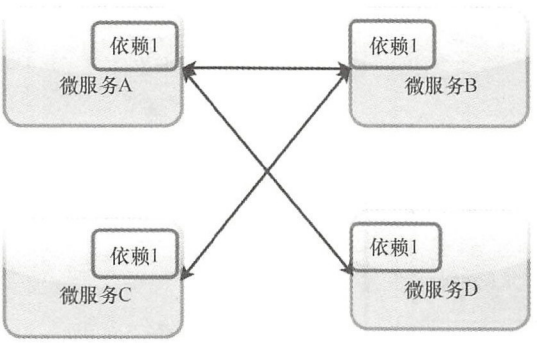


图 10-3

第二种情况是，如果两项或多项服务中存在相同的依赖关系，应该怎么去处理呢？如果这个依赖关系出现更新，那么应该以怎样的顺序来更新具有相同依赖关系的服务呢？微服务彼此之间的交互行为很可能导致一个微服务的更新引起其他微服务的调用失败。

10.1.4　服务部署数量与日俱增

微服务架构是将一项服务分解成多个单独的微服务组件。多个的微服务组件将会使用多个不同的代码库，从而每个微服务组件的部署周期也不尽相同，使得日常的部署频率也大大增加。这种连锁反应还会加重基础架构的运营负担，例如，必须保持部署速度，

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

为每一个微服务创建单独的部署流水线，以及确保部署过程中微服务之间不会互相影响，等等。如果有超过 20 个微服务需要与新构建的微服务进行通信，那么，测试所有这些微服务将花费大量的时间。这时就需要自动化测试的解决方案，而此类解决方案又会提出新的资源需求，如测试环境的搭建等。与单体应用相比，微服务的运行还需要显著增加服务器数量，而服务器数量的增加意味着运营成本的上升。组织中对 DevOps 的依赖或投资，是由于组织中环境或服务器数量的增加。因此，不断上升的 DevOps 成本也是需要考虑的问题。

10.1.5 监控多项服务，发现性能下降或其他问题

正如 10.1.4 所述，如果部署的微服务数量与日俱增，那么监控这些微服务将成为另一个严峻的挑战。有时候，DevOps 团队无法维护也无法搞清楚所有类型的监控数据。也许，DevOps 团队可以使用仪表板来显示有关服务器及其进程的各类参数，如 CPU 使用率、磁盘空间以及网络流量的大小等。但是，维护、理解和应对如此众多的信息并不容易。新部署的服务可能无法被另一台设备的网络所覆盖。因此，尽管部署顺利完成，但这些服务组件之间可能无法正常通信。通过监控工具就可以知道某个微服务以及这个微服务所在的服务器是否可用，但却很难监测到每个服务是否能访问需要的其他微服务。监控还会导致服务更加细分，最终可能无法作为一个整体来支持整个系统。面对如此众多的微服务组件，其挑战便是如何借助适当的工具对这些微服务进行集中化的监控。

10.1.6 日志与不同组件的关系

微服务架构中存在很多的服务组件，既可运行在相同的实例上，也可运行在不同的实例上。正如上文提到的那样，获得有关微服务架构日志的完整视图是非常困难的，所以就很难了解某些特殊请求的具体处理情况。当出现问题时，必须有人进行调试，但此人无须熟知平台中使用的所有技术，也不必知道各项微服务的实时情况。他们只需检查并解析日志记录，并从中了解到整个系统的实时概况即可。此时，他们将会遇到另一个问题，即使所有的日志均存放在同一个地方，也很难从庞大的数据中提取有用的信息。由于不同的服务组件是由不同的开发者使用不同的语言开发的，并且在开发时未能遵循任何统一标准，因此，不同服务组件采用不同日志格式的概率极大。以 Java 样例日志格式输出为例，如下所示。

```
2017-05-26 15:34:53.787 INFO 13262 --- [io-10000-exec-2] i.s.c.controller.
< Controller Class> : <Log content>
```

这行日志中有日期、时间、日志级别、类名和日志内容。Python 的标准日志记录格

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

式不同于 Java，Ruby 亦是如此。简而言之，不同的语言具有不同的日志记录标准和不同的日志库，所以不推荐使用正则表达式之类的工具来查找不同日志间的内容。

10.2　常见问题的解决方法

10.1 节主要探讨了在微服务架构中遇到的一些常见问题。本节将尽力找出这些问题的解决方案。

10.2.1　解决性能问题的步骤

微服务的性能不仅取决于代码，还取决于网络延迟、消息序列化/反序列化以及应用中的动态负载等因素。这实际上增加了性能测试的范围。因此在解决性能问题时，添加资源是第一步举措。当在垂直方向上扩展应用时，向服务器中添加资源只会带来一定程度的性能增益，添加更多的资源并不能总是获益。所以第二步可以考虑在上一步的基础上添加负载均衡器，例如，在前端添加负载均衡器，并在后端添加更多的服务器来应对流量的增加。当流量很高并且服务器数量增加的情况下，还需要对流量进行优化处理。我们需要仔细观察旧数据的模式特征并据此做出合理的自动化规划，来对流量进行预测并适时添加新的服务器来处理增加的流量。

任何根本不响应请求或响应速度极慢的服务器均应停用。例如需要 5 秒应答请求的应用，其他服务只需要几毫秒来应答。负载均衡可利用多项技术来实现。对于互联网上的数据而言，利用轮询（round-robin）算法来实现负载均衡可达到 95%的请求成功率，而利用最小负载服务器（least loaded server）的负载均衡算法则更加智能，可将请求的成功率提升至 99%。

除了高流量问题外，微服务的代码也可能存在问题。例如，客户端的一个 API 调用可能会触发多个微服务之间彼此通信，那么这将很难找到响应时间较长而引起性能问题的那个微服务。某些工具可以给出性能的衡量指标。例如，可以在应用前面部署一个 Nginx 或 Apache 等 Web 服务器来记录响应时间，并对响应时间编写警报。当响应时间超过阈值时，就可以添加一台新的服务器并将响应速度缓慢的服务器从负载均衡器中移除。此外，还可以对每项微服务单独进行性能监控。Graphite、Statsd 或者云服务应用 New Relic 等，均提供丰富的可视化仪表板来帮助发现问题。有关这些仪表板的具体信息，参见第 9 章的讨论。

10.2.2　处理不同语言生成的并处于不同位置的日志记录

查看日志始终都是开展调试工作的第一步。由于系统中运行着大量的微服务，如果

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

系统出现问题，开发人员便需要深入研究这些日志记录。但是，由于不同服务组件拥有的日志文件太多，从这些海量数据中找到一条调用的踪迹并不容易而且相当耗费时间。解决这个问题的办法是使用日志聚合工具，例如将开源的日志聚合工具 Logstash 和 Kibana 联合使用或使用云服务日志聚合工具 Loggly。Logstash 可将日志文件聚合到一个位置，而 Kibana 则可在日志中进行搜索。上述两个工具均已在第 9 章中进行了介绍。但问题并不总是出现在服务的组件之中，也有可能出现在辅助工具之中。就像运行在 JVM 中的 Java 服务一样，如果用户给 JVM 分配的内存太小，JVM 将会频繁地进行垃圾回收，从而拖慢服务组件的处理速度。

日志格式是需要解决的另一个问题。所有服务组件均应采用统一格式。一般情况下，所有的组件均可通过明确定义的 JSON 格式开始发布日志，使得查询日志变得更加方便。这更像是使用 JSON 格式将数据存储在 Elasticsearch 之中，从而实现极为快速的数据查询。作为基于云的日志聚合工具，Loggly 可以支持基于 JSON 的日志查询，同时还提供设计精良的界面，允许用户按标签或某个时段等标准来显示日志。

即使日志可以汇总在一起，但仍然无法解决在平台内部追踪客户端调用的问题。如果一个微服务组件向下一个微服务组件发出 API 调用请求，下一个微服务组件在接到请求之后又将其传递给再下一个微服务组件，如此传递下去，就很难了解这个调用请求在接下来的微服务组件中会发生什么。那么开发人员如何才能弄清楚某个特殊的调用请求及其处理过程呢？首先是给每个调用平台绑定一个唯一的 ID，让这个 ID 绑定在调用请求中，并一同传递过来，从而跟踪每一个调用请求。微服务在处理新请求时，必须为其生成并记录下这个 ID，这需要编写一定的代码，然后再将其传递给另一个微服务。借助这个 ID，我们能够洞悉某个特殊请求的整体情况，为调试工作奠定了坚实的基础。

Microsoft SharePoint 等许多商业软件包都在使用该模式来帮助解决问题。

10.2.3 服务之间的依赖关系

10.1.3 节曾提到了位于不同微服务中的通用库把微服务的技术栈限制在某种语言之中。为解决这个问题，使微服务之间彼此完全隔离，可以使用代码冗余，即代码复制来解决不同微服务间的依赖问题。但这个方法只是一个折中方案，并非最佳解决方案。随着微服务架构的逐渐演变，可能会从实践中迸发出更好的想法。此外，在微服务架构中，对任何依赖关系所做的任何改变都绝对不能打破向后兼容性，除非所有的服务均已升级到最新的版本。

10.2.4 DevOps 专家积极参与

面对如此众多的微服务环境，以及运行在平台之中大量的服务器资源，全天候待命

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的 DevOps 专家成为必需的资源。如果有人希望发布一个微服务，就需要 DevOps 专家来规划产品发布和部署事宜。在单体架构中，任何发布都是一件大事，可能需要持续几个月才能完成。任何失败都将导致整个平台宕机并导致发布时间延期。因此，DevOps 专家始终必不可少。一般可以通过两种方法来处理这个问题。

1. 使用专用的工具

借助 Jenkins、Travis、Chef 和 Puppet 等专用工具以及封装在这些工具之中的某些自定义脚本即可实现全自动的微服务部署。首选是实现一键式部署。

2. 培养聪明的开发人员

开发人员也需要去学习 DevOps 技术。开发人员无须了解 DevOps 的全部内容，但在朝着微服务架构进行迁移时，他们应将自己想象成为微服务的拥有者，其责任不仅限于编码，还需要了解从迁移开始直到微服务投入生产运行之间的每一步行动需求。

结合使用这两种方法，DevOps 工作可以在一定程度上分派给多人并行执行。响应式监控有助于践行 DevOps。10.2.5 节将会讨论这个问题。把应用部署在云上而不是独立服务器上，会大大简化 DevOps 的运维工作。弹性伸缩是云计算的最大优势，可以通过制定不同的规则来进行弹性伸缩，这有助于整个应用及时地处理突发性的流量增长。例如，对于任何服务的响应时间增加至 400 ms，就需要部署更多的服务器。

10.2.5　监控

把一个庞大的系统分解成多个子系统并分配给多人负责时，将很难监控每个子系统是否都在正常工作。为了解决这个问题，首先需要将整个监控系统进行集中化处理。无论单独运行了多少个微服务组件，都应实施集中式日志记录与监控管理，以便加快调试速度并提高调试效率。

接下来是响应式监控或综合监控。此类监控应该同时工作于微服务内部和微服务与微服务之间。如果某个微服务故障，某些服务停止与数据库通信，或某个服务遇到磁盘空间等问题，监控系统必须先于终端用户了解到这种情况。这也正是需要实施响应式监控的原因。即使监控工具不能修复问题，至少可以尽快通知相应的团队进行处理。第 9 章介绍了多种监控工具。除了这些工具外，还可选用 Zipkin，用于调试分布式应用。

Zipkin 是 Twitter 用于帮助开发者跟踪和调试微服务的开源工具，设计灵感源自 Google 的 Dapper Idea。Zipkin 依赖于 Apache ZooKeeper，可将用于追踪延迟问题的时间数据汇总起来，并给每个请求自动分配唯一标识符。如果想启用 Zipkin，每个微服务均需装载

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Zipkin 库，以帮助服务识别出请求的入口点。

10.3 小结

本章讨论了一些常见的微服务问题以及如何解决这些问题，其中的大部分问题均可通过适当的工具得以解决。而其余的问题则需要摸索着加以解决。希望本书能就如何在实际环境中使用微服务为读者提供一些有用的信息。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

微服务实践

目前微服务架构在软件开发领域得到了广泛应用，因为它可以将一个大规模的应用分而治之，从而分解成许多较小规模的微服务，这样在减少系统内部各组件依赖的同时，还可以让整个系统易于维护和扩展。采用微服务架构的应用可以很容易地修复其中某个微服务的故障，从而消除系统彻底宕机的可能性。本书使用Java语言，并以实际的样例来说明如何快速高效地构建和实施可扩展的微服务架构。

本书以微服务架构的简要介绍为切入点，将微服务架构与其他架构进行了相关比较，详细探讨了微服务组件的重要特点和微服务端点之间的通信方式，以及如何建立有效的数据模型，如何对微服务进行测试和快速部署，然后总结了软件应用从单体架构迁移至微服务架构的最佳实践，最后着眼于微服务架构的监控、扩展和故障排除，可以为开发人员从头实现微服务架构打下坚实的基础。

读者将学到以下内容：

- 了解服务发现和外部配置在微服务整体架构中所扮演的角色；
- 学习如何在事件驱动的微服务中使用消息代理；
- 学习如何在不同组件之间混合使用各种数据管理策略；
- 基于Spring Boot的开发环境对微服务进行多种类型的测试；
- 将持续集成应用于微服务架构中；
- 使用多种工具来监控和扩展自己的微服务应用。

异步社区 www.epubit.com

异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-49870-0

ISBN 978-7-115-49870-0
定价：49.00元

仅供非商业用途或交流学习使用